A Framework for Building Collaborative Applications

YC Nuckchady

University of Tampere
Department of Computer Sciences
Computer Science
MSc Thesis
November 2004

University of Tampere
Department of Computer Sciences
Computer Science
YC Nuckchady: A Framework for Building Collaborative Applications
MSc Thesis, 75 pages
November 2004

Abstract

The advent of the Internet has been driven by the relentless effort of decades of research to develop a digital environment that would facilitate collaborative work on a global scale. The popularity of the Internet soared when the world saw the potential of the World Wide Web (WWW) as an Internet-based tool for sharing digital media and communicating. The success of the WWW has been so tremendous that most of the effort invested in the development of Internet-based tools are focused on improving the WWW rather than investigating other forms of collaboration. Another explanation behind why there are so few collaborative applications is because it can be quite complicated to design and develop them. To this end, we propose in this thesis a framework written in Java that would facilitate the development of collaborative tools. It does so by providing ready-made mechanisms such as, concurrency control which is common in every such applications.

This study undertakes a constructive approach by identifying those features of Java such as, object serialization, that is exploited in our design of the framework. We also analyse several forms of socket-based communication which is essential for networked applications. Java's Remote Method Invocation architecture is also investigated to get a sophisticated insight into how our collaborative model should operate. Finally, we conclude our analysis by studying the Java Messaging Service to obtain an alternative view on synchronous and asynchronous communication models.

The scafolding of our collaboration model is based on the concept of an *island* which is essentially a star topology with a central coordinator and a number of satellite participants. It is demonstrated that this arrangement provide a robust concurrency control mechanism. Islands can be divided into smaller islands as a mean to counter the problem of overload which is peculiar to collaborative applications. The islands are interconnected via their coordinators to form a full-mesh topology and thus provide a very stable and scalable architecture.

In order to gauge the effectiveness of our framework, we enhance a typical single user application to a collaborative one using the toolkit from our framework. It is shown that in extreme cases, when the applications generate large amount of data, our solution can easily withstand the creation islands of 11 users and after that it would be necessary to split the island. And under *normal* circumstances, islands can easily accomodate between 25 to 30 participants at a time before the load balancing mechanism has to be invoked.

Table of Contents

Acknowledgements

From the moment I started this research, this thesis became for the many months to come, the lighthouse that guided me through life during that period. Writing the thesis has been a sea-adventure in all sense: moments of joy when my clouded ideas cleared off, frustrations when I could not see where I was going and sheer excitements when I discovered new routes to my lighthouse. But even though I had a direction, it would not have been possible if I have not been able to lean on the shoulders of the giants that were always around me. So, I would like to express my deepest and most sincere gratitude to the following people: To my dear friend and supervisor, Professor Jyrki Nummenmaa, to whom I am eternally grateful for having given me this once in a lifetime opportunity and who always pointed me towards my lighthouse. To my *kulta*, Päivi, for being the rock I could tie my boat when I was exhausted and for being the source of energy when she would greet me with a smile. To Ruska, Bono and Sfinxi for being tirelessly playful and always providing me with a soothing atmosphere. To my father and mother, Pranesh and Mila, for teaching me about life and for being my spiritual light. To my other Professor, Erkki Mäkinen whose invaluable assistance has helped me polish this work to its final state. Lastly but not the least, to all my friends who were helping me even though they were not aware of it.

To all of you again: **Thank You !**

1 Introduction

Collaboration has been in existence ever since people started living in communities. Some say that it is the glue and energy that maintains a society together and drives it forward. Early forms of collaboration can be traced back to the stone ages, when people worked together to hunt down large animals or defend their territory against invaders. Throughout history, we have witnessed that technological innovations like, railroads, planes or radio waves have consistently improved the collaboration effort by shortening the time required for distant parties to communicate. Similarly, the advent of Internet has brought with it even faster and inexpensive means of communication such as, IP telephony, email or instant messengers. More importantly, it has created communities where geographical separation and borders have no meaning. These virtual societies group people based on their particular interests rather than their physical neighbourhood. The Internet has indeed raised the process of collaboration to its ultimate level as it facilitates the creation of concentrations of shared interests, which is after all the motivation behind why people collaborate.

1.1 History of Collaborative Applications

As a matter of fact, the initial drive towards the creation of the Internet has been the desire by physicists to access remote supercomputers in order to collaborate on solving complex mathematical problems [Abbate, 2000]. Since the 1960's, researchers and engineers have been working relentlessly to formalise and perfectionate the process of plugging a computer into a network composed of a wide variety of devices interoperating harmoniously [Leiner *et al.*, 2003]. Hence, towards the end of the 1980's, the Internet had evolved to a satisfactory level with regards to the protocols required for communication. This prompted the creation of indexing services such as, Gopher and WAIS which would aid Internauts (Internet users) in locating resources on the global network. In 1989, Tim Berners-Lee, a physicist at CERN, invented a highly sophisticated indexing service called the World Wide Web (WWW or Web) which would allow the physics community to collaborate by sharing their research work [Wikipedia, 2004]. Since then, seduced by the amazing power of the WWW, other communities such as, the pornographic industry, have helped to make the WWW grow by adding more servers on the Internet. This viral effect stole the attention of researchers and industry leaders away from devising new collaborative tools for the Internet. Indeed, it seems

that, taken by the commercial success of the Minitel, a videotex service that offered tools such as emailing and online booking [Abbate, 2000], efforts have been directed towards forging the WWW into a similar business model. Fortunately, the video game industry also realised the potential of the Internet and in 1994, ID Software released DOOM and 3D-shooter game [Cass and Kushner, 2002]. The hightech computer graphics of the time coupled with convenient modem play made it an instant success. Multiplayer capabilities became a standard feature in subsequent games by ID Software and was picked up by others in the industry. The latest milestone in the history of Internet collaboration occured towards the end of year 2000 , when Shawn Fanning created Napster, a peer-to-peer (P2P) file sharing application. Unlike for example, the WWW or File Transfer Protocol (FTP) which are strict client-to-server forms of operating, P2P makes no distinction between the two, rather a computer is known as a node. This means that even the slowest node with the slowest connection bandwidth can serve files and be a client to other nodes at the same time [Balakrishnan *et al.*, 2003]. The P2P approach, meant that there are more servers on the Internet than ever before, and this implied that shared resources are more easily available and accessible. Also, unlike an FTP client which operates by downloading a file from one server, many P2P applications supported multiple downloading. More explicitly, since multiple copies of a file are accessible on several nodes, the P2P application downloads a part of the file from each node simultaneously and therefore obtains the document faster than with the FTP method. This ablility contributed to the lightning fast growth of their popularity. Programs like Napster, Kazaa or BitTorent dominate the P2P market because they are mainly used for downloading entertainment media. However, there are other types of P2P applications such as Seti@Home[1] which performs distributed computing by making the volunteered computers to perform some specific calculations when they would otherwise have been idling. In fairness, it must be stressed that P2P applications existed before Napster [Hayes, 1998].

1.2 Current State of Collaborative Applications

To date, there are over 8,000 official network services that operate on the Internet and many more can be built by using the non-reserved port numbers in the range of 49152 and 65535 [Authority, 2004]. Yet, very few of these services as for

[1] `http://setiathome.ssl.berkeley.edu/`

example, the HyperText Transfer Protocol (HTTP) or FTP can be considered as collaborative in nature. The current trend adopted by many industry leaders such as, Lotus and Netscape, has been to gather these popular services into a suite of collaborative tools, commonly known as a *groupware*, with the prospect to engage the users into an adequate collaborative experience. According to Brinck [1998], a groupware is a collection of synchronous and/or asynchronous tools, where, the latter are intended for situations where it is not necessary for the user to collaborate at the same time, while in the former case, it is imperative that they do so. In the above example, when P2P is used for sharing files, those applications are said to be asynchronous, because in this mode of operation, a node can stop downloading a file or interrupt an upload at any time and it can resume the process at a later stage. Seti@Home is an example of a synchronous P2P application. Examples of synchronous groupwares include shared whiteboards, video conferencing, chat systems, decision support systems and multi-player games. Email, newsgroups, mailing lists, group calendars and collaborative writing systems are examples of asynchronous groupware [Orfali *et al.*, 1999]. Fig. 1.1 compares the synchronous and asynchronous communication modes.

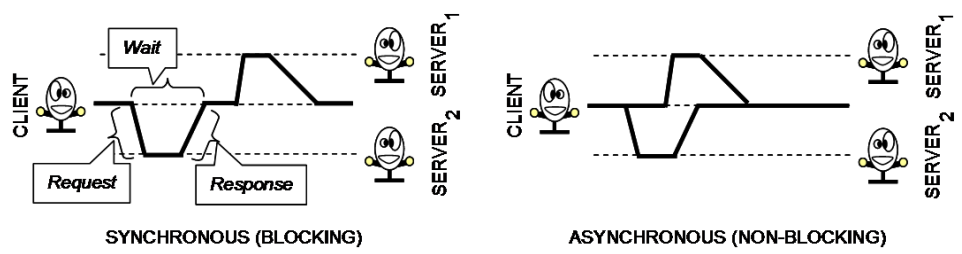

Figure 1.1 Comparison between the synchronous and the asynchronous communication mode. The solid lines indicated period of activity by the associated computer while the dashed one represents inactivity. The left-to-right direction of the lines expresses the flow of time.

Our main criticism towards groupwares is that there are too few applications that provides the collaborators with hands-on interaction. In other terms, in multi-player games, such as DOOM, the players can control their respective characters and influence the game through an interactive terminal. Similarly, in the case of the whiteboard application, all collaborators can freely manipulate the texts and drawings on the board. But these are exceptional cases where the collaborator can exert a direct effect on the object of the collaboration effort. The typical way of proceeding in most collaboration situations is that the collab-

orators communicate what they want to be done or their ideas or decisions via assistive tools such as emails, online forums or instant messengers. This indirect approach to collaboration can be quite slow because often there are too many misunderstandings among the involved parties. This can be accounted by the fact that the object on which they are collaborating is isolated from the discussion process. Microsoft has addressed this particular aspect of collaboration. It came up with Netmeeting, a tool that provides application and desktop sharing in addition to the above mentioned asynchronous groupwares [Joyce and Moon, 2000]. As an example, consider a collaborative session aiming at an end-of-year report. Now, one of the collaborators can make the documents available to everybody by opening them in Word and Excel, and sharing the latter via Netmeeting, and still pursue the discussion via say, video conference. This way all the interested parties can view the object through these discussions and make precise comments or coorperate by taking turns to fix the various disagreements using the application sharing capabilities. Netmeeting also provides an application development framework that can be used to create new third party softwares that can plug into the latter and become shareable. The main drawback with Netmeeting is that it is based on ActiveX controls and can therefore only run on Windows-based platforms. Moreover, Netmeeting is expensive and is therefore not accessible to everybody. However, the financial hurdle behind Netmeeting is not the only reason of why there are so few collaborative applications with data sharing capabilities. One reality about these types of applications is their complicated architecture as they try to address the various sophisticated features of their operational usage. For example, in the case of the whiteboard application, the fact that several users can interact with an object at the same time, demands that the collaborative system must guard against those interactions that can violate the integrity of the object and still keep the response times of these interactions within a reasonable time frame. So, for such types of applications, the developers have to consider numerous extra design features like, concurrency control or excessive processing loads. These extra issues can turn out to be a discouraging factor, as they divert the attention of the developers away from the actual features of the applications, which are the reasons why these developers wanted to create such applications.

1.3 Structure of the Thesis

In this thesis, we propose a framework and a toolkit, called the Collaborative Toolkit (CTK), that would aid in the development of collaborative applications

with data sharing capabilities, by providing built-in mechanisms for concurrency control, load balancing and synchronisation strategies. Hopefully, our proposed architecture should help developers to focus on the first order features of their applications and thus create quality products. Moreover, CTK is written entirely in Java, which therefore allows it to run on a very large number of platforms. The thesis is structured as follows. In Chapter 2, we describe the general idea of designing collaborative applications using Java. We also introduce some of the advantages that the latter offers and which we exploit in the design of CTK. Furthermore, we analyse two different programming models based on Java, Remote Method Invocation (RMI) and Java Messaging Service (JMS), with the aim to understand how collaborative applications built with them operate. This study also describes the technical details that these technologies employ to achieve collaboration. Lastly, we criticise and identify the useful features that can be used in the design of CTK. In Chapter 3, we give a formal definition of what we mean by collaboration and which we use to guide the design of our framework and toolkit. A qualitative study of various network topologies is also carried out in the scope of identifying the best model for our needs. Towards the end of the chapter, we discuss the three main concerns that always arise in every collaborative system: concurrent access to data, performance bottleneck and synchronisation. We address each of them by proposing the solutions that CTK implements. Chapter 4 explains in detail the adopted collaboration model that all applications built using CTK should follow. In the software architecture section, we provide technical details about the various modules that enable collaboration to take place in CTK-based applications. In Chapter 5, we introduce a single user application that has been given multiuser capabilities through CTK and we propose some experiments based on it in order to evaluate the toolkit and framework from an empirical point of view. Finally, Appendix A, details the underlying mechanics of Java's implementation of threads. This study was useful in desiging the concurrency control and synchronisation strategies implemented by our framework.

2 Background Study

A collaborative system written in Java is essentially a collection of connected *Java Runtime Environments (JREs)* which interoperate by exchanging data over a network as illustrated in Fig. 2.1. Java does allow a large array of devices (in terms of hardware and underlying Operating System (OS)) to work harmoniously. When an application needs to send data, it invokes some specific objects running in the layer below it, the JRE. These objects in turn make appropriate system calls provided by the host OS to convert the data into a sequence of bits [Naughton and Schildt, 1999]. These bits are then taken by the underlying hardware and sent as signals (electrical or radio waves) on the Internet. When these signals are received by the intended device, the OS translates the bits produced by the hardware layer into a sequence of bytes. These are then handed over to the JRE which processes them further into a meaningful format before forwarding the result to the listening collaborative application. There are a number of programming models in Java that can be employed to build collaborative applications and in this chapter we study two of them, RMI and JMS. Together they cover a large number of aspects pertaining to collaborative applications. For example, RMI is best suited to design applications for the field of distributed computing (e.g., cracking of encryption keys) while JMS is ideal for developing asynchronous applications such as group calendars. Finally, we discuss socket programming which is central to many networking applications and every Java based technologies including RMI and JMS.

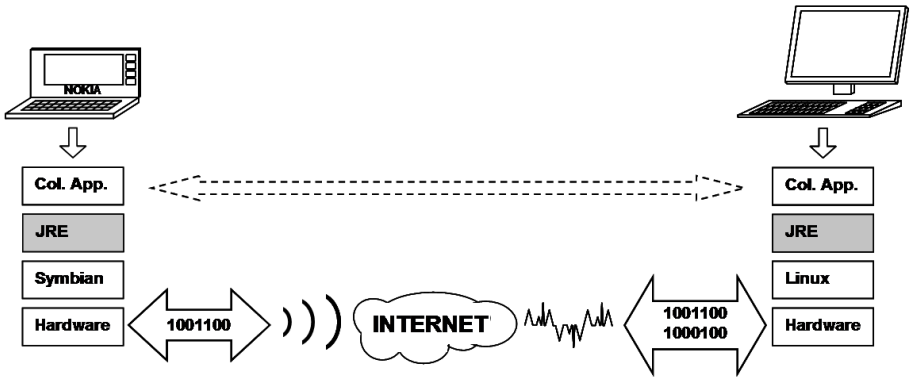

Figure 2.1 This core representation of all collaborative systems built using Java. The dashed arrow represents the virtual bi-directional link that the applications *use* to communicate with each other.

2.1 Java as a Programming Solution

Compiling Java source files results in object files called *class* files which are made up of bytecodes. These bytecodes are in an intermediary form between the language that the programmers use to the write the application and the code that the target computer understands and uses to run the program bytecodes. Unlike some other programming languages such as, C or PASCAL, which are also compiled, Java does not run directly on the target computer but rather on a virtual computer known as the Java Virtual Machine (JVM) that fits on top of the former one. The purpose of the JVM as a machine is to implement a specification known as the *Java Virtual Machine specification*[1] that describes how it should execute the bytecodes on the host computer. The specification is flexible enough to allow vendors of JVMs to employ their optimal implementation techniques, so that their JMV can perform to the highest standards on the target operating systems they are aiming for. Moreover, the specification is so flexible that it even allows the JVM to be implemented partially in terms of hardware [Venners, 1998]. The great flexibility of the specification allows Java programs to run on multiple platforms such as those understanding X86 or RISC instruction codes. The JVM specification is closely tied to the Java Language Specification (JLS)[2] and Java compilers have to implement the rules in the latter with upmost strictness in order to generate legitimate bytecodes. Therefore, it does not matter on which platform a Java program has been developed as it is guaranteed to run on any computers or devices for which a JVM exist. Of course, if the bytecodes have been produced for a 32-bit computer, they will run never run on a 64-bit based architecture and vice-versa. However, the portability is not so straightforward, as sometimes finetuning is necessary in order to make the programs run properly. For example, the memory footprint of a thread on a Linux based computer is smaller than that on a Windows based computer. Therefore if an application creates a fair number threads, then it will tend to run slower on the Windows platform than on the Linux one assuming that both have the same hardware configurations. The JVM is available through the JRE, which is the execution space of all Java programs. The JRE also provides a number of core libraries such as, java.io or java.net that Java application developers can access via the Application Programming Interface (API) [Venners, 1998]. These core libraries

[1] `http://java.sun.com/docs/books/vmspec/`

[2] `http://java.sun.com/docs/books/jls/`

are mainly used to carry out host-specific tasks like, creating files, sending data on the network, or accessing the windowing system. The JRE is a pluggable architecture that allows *extensions*, which are packages of .class files to be added to the collection of core libraries. The purpose is to add to or improve on the set of core features provided by the underlying system [Sun, 1999].

It can be argued that for example, C or C++ programs can also be compiled for the various target computers and thus achieve the same cross-platform abilities of Java. But the main problem is that there are many standards (ANSI, POSIX) that are used to develop specifications for these languages and they conflict each other in some aspects. For example, UnixLib is used to make UNIX based programs run on RISC OS but does not conform with the POSIX standard which all UNIX systems are based on [Naulls, 2004]. Indeed, with Java, there can be different JVMs but they all implement one specification of the language.

When dealing with different computers as it is often the case in collaborative systems, it is very important to take into consideration the order in which multibytes data are locally stored. There are two ways of organising data in a computer, the *little endian* and *big endian* formats. In the former case, the smallest order byte is stored first and at the lowest available address and the next byte in the next lowest available memory block and so on and so forth. In the other case, the order is the other way around. That is, the highest order byte is stored first and in the lowest available memory block. Intel based computers use the little-endian format while Motorola processors found in Apple Macintosh computers use a big endian byte-order scheme [Green, 2004b]. Fig. 2.2 illustrates the concept behind byte-order and the problem that ensues when transfering data straight between two computers with different endian formats. The data is stored on the target computer in the same order as it was stored on the originating computer. Therefore the copied data will be clearly misinterpreted when it is read. In the field of networking, where this problem is a permanent issue, the common approach is to convert data that will be transmitted from the host's byte-order to the network byte-order (which is big-endian) and back to the destination computer's byte order when it is received. This programming practice ensures that this issue never becomes a problem. Java programmers do not have to worry about this data conversion process as it is automatically carried out by the JVM during the process of *serialization*. This automated step that is always carried out when data is written to an OutputStream object or read from an InputStream object [Grosso, 2001].

In every Java application, objects go through a life cycle of loading, linking,

initializing and finally unloading (or garbage collecting) [Venners, 1998]. The loading process is performed by a specialized module called the *class loader* which can be the default that comes with the JVM, or it can be a customized one where the user has defined new policies. For example, a policy can be to download class files from a particular network point instead from the CLASSPATH property of the host operating system. The role of the class loader is to read the bytecodes from a .class files and create an instance (an object) of type java.lang.Class which represents that class at runtime [Liang and Bracha, 1998]. In Java, classes are loaded whenever they are needed as it is demonstrated by the simple experiment in Table 2.1. When the program is run with a statement like java First, the JVM creates a main thread that tries to execute the codes enclosed in the method public static void main(String[] args). But the thread notices there is no binary representation of First.class in the JVM and therefore uses one of the system's class loaders to load that class. At this point, the debug output shows Loaded First which confirms the success of this process. The main thread then enters the main method and executes the first statement which causes First created to be printed on the screen. Next, the main thread encounters the statement Second s = new Second(), and for the first time it comes across the symbolic reference to the class Second, s. It therefore calls on a class loader to load the definition of that class in the JVM just like previously and this process is punctuated by the line Loaded Second printed to the debug stream. Note that, once the classes are loaded, they are automatically linked and initialized by the JVM so that they can take part in the execution of the program [Gosling *et al.*, 1996].

The Java programming language offers in its Application Programming Inter-

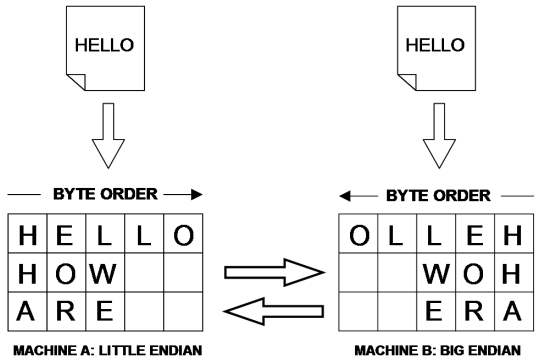

Figure 2.2 String "HELLO" is stored locally on two computers with different endian formats and is transferred from one to the other without a proper control.

face (API) the same classes that are used by the JVM in the process of dynamic class loading [Chan *et al.*, 1998, Gosling *et al.*, 1996]. In addition, it also provides a mechanism called *reflection* that enables a program to investigate the characteristics of a class during runtime. One of the ways of proceeding is via the static method forName() from the class java.lang.Class, which takes as argument the fully qualified name of the class to be investigated. After clearing the access with the local security manager, the JVM dynamically loads that class and makes it available in much the same way as when a user would open the corresponding source file in an editor and reads its content. At this point, instances of the class can be created by selecting the most appropriate constructor for the circumstance. In a similar fashion, the public methods can also be invoked and parameters passed to them [Green, 2004a]. Indeed, reflection coupled with dynamic class loading is a very powerful tool that very few programming languages (e.g, Microsoft's .NET framework) can boast to provide. However, one must note that the process of loading classes at runtime and creating objects takes a very long time and can be detrimental to efficiency of the application that makes use of it.

```
public class First {
    public static void main(String[] args) {
        System.out.println("First created");
        Second s = new Second();
        System.out.println("Second created");
        s.execute();
    }
}
...
public class Second {
    public void execute() {
        System.out.println("Second Executing");
    }
}
```

```
[Opened c:\jdk\jre\lib\rt.jar]
...
[Loaded java.lang.Object ...]
[Loaded java.io.Serializable ...]
[Loaded java.lang.Comparable ...]
...
[Loaded java.lang.String ...]
[Loaded java.lang.Class ...]
...
[Loaded First]
First created
[Loaded Second]
Second created
Second executing
```

Table 2.1: The code listing on left demonstrates dynamic class loading while the table on the right is the proof from the debug output obtained when the program First ran with the -verbose option.

2.2 Socket Programming

Generally speaking, a socket in an abstraction of the Input/Output (I/O) device that connects a computer to a network. In the early 1970's, UNIX computers were designed to serve as timesharing systems and to this end, the required network I/O operations were modelled according to the I/O operations of the UNIX file system. So for this historical reason, socket programming follows the paradigm of *open-read-write-close*. That is, when data is sent or received between two computers, a socket is opened, data is read or written and the socket is finally closed, just as one would proceed when dealing with a file [Comer, 1991]. So, in the context of collaborative applications built using Java, the programming practice is to create a Socket object which opens the connection to the other computer. Then, from this socket, two data streams are extracted: an input and an output stream for reading and writing purposes as in the case of bi-directional communication. When the application wants to end the communication, it closes the socket which translates to closing down the two data streams [Harold, 2000]. Java offers a variety of I/O streams with specific built-in mechanisms for handling different types of data (e.g., binary, ascii, etc) or perform special tasks (e.g. CipherInputStream reads and automatically decrypts data from the network before handing it over to the module that needs it) [3].

To help understand the client-server programming model in Java, consider a typical situation where a user wishes to read a webpage. In this case, our two communicating computers, the client and the server, are the web browser (browser) and the web server (server) hosting that document. When the user types the name of the document in the browser, he or she also specifies the address of the server in the form of a domain name (e.g., www.yahoo.com) or the equivalent IP address (in this case, 216.109.117.106). In the context of Java programming, the server is basically a special socket, called a *server socket*, and is implemented by the ServerSocket object. The latter sits on a *port* (e.g., 80 or 8080), which is a logical abstraction of the network medium, and listens to incoming connections just like from the browser. Indeed, when the latter indicates that it has connected to the web site, what actually happened is that , it has created a socket and established a connection to the server socket. At which point, the latter creates a temporary socket and binds the incoming connection from the browser to that new socket. This way, the server socket frees itself to accept

[3] Consult the Java API (Ver 1.4.2) for definitions of javax.crypto.CipherInputStream and javax.crypto.CipherOutputStream.

new connection requests from other potential browsers. In the next step, both applications extract a pair of I/O streams from their respective socket so that they can communicate (send and receive data). It should be clear at this point that, whatever the browser writes in its output stream will come out of the input stream at the server's side and vice versa. This type of communication is known as a point-to-point approach and coincides to the virtual connection illustrated in Fig. 2.1. In this example of a client-server application, the browser and the server can from now onwards engage into HTTP for exchanging data between them. For example, the browser makes a `HTTP GET` request to the server for the particular document and if the latter is in the public domain of the server, it is converted to bytes and streamed down to the browser. Otherwise, the server issues an HTTP 404 response to the browser. In both cases, the browser interpretes the byte streams, reconstructs the information into a readable format and presents it in its viewport to the user. When the latter decides to visit another website in that browser window, the browser application closes the socket to the previous web server, opens a new socket to the new one and repeats the process.

2.2.1 TCP and UDP Sockets

As suggested above, network applications can be differentiated according to the types of protocols they employ but they can also be grouped as well by using the protocols as a criterion. For example, Windows Explorer and Internet Explorer are well known file browsers and they both employ the Internet Protocol (IP) to communicate on the Internet. However, they differ at a finer level of more specific protocols: the former uses FTP to manage files on remote computers while the latter employs HTTP to achieve the same. As a matter of fact, the International Standards Organisation (ISO) provides a structure called the Open Systems Interconnection (OSI) that can be used to describe a network application in terms of the protocols it uses. The OSI model is made up of seven layers, where each of them specifies how data is exchanged between two applications on that particular layer [Tanenbaum and van Steen, 2002]. For example, in the case of Internet Explorer, at the *network* layer, it abides to the IP to exchange data with a web server, while with regards to the *session* layer it communicates using HTTP. So, based on the OSI model, network applications can be viewed as a stack of seven protocols describing how data is exchanged between the corresponding layers. It is possible for an application to use more than one protocol at a layer so that it can adapt to various needs. For example, if a groupware is considered

as an application unit, then it can employ SMTP, FTP and HTTP at the session layer of the OSI model to send mail, backup files and browse the WWW.

Since the type of collaborative application that concerns us will run on the Internet, they will all be governed by IP at the network layer of the OSI model. The Java programming environment provides the implementations of two protocols that can be used at the *transport* layer: Transmission Control Protocol (TCP) and Universal Datagram Protocol (UDP). Consequently, network applications in Java can be TCP/IP (which means TCP over IP) or UDP/IP according to the OSI model. TCP has been designed for a connection-oriented mode of communication, while UDP is to be used strictly with applications that need a connection-less type of communication. The former model of communication can be compared to the process of making a phone call. First, the caller dials the number of its correspondent and waits until recipient picks up its phone. Next they introduce themselves and only then they engage into the conversation. At the end of the conversation, they can exchange salutations to formally acknowledge its end and the call terminates when one of them hangs its phone [Chow and Jonhson, 1998]. On the other hand, the connection-less mode is much simpler and analogous to the process of sending a long sms. Here, the sender starts writing the message and at end selects the phone number of the recipient and send the message. The phone's built-in computer then splits the message into portions of length less than or equal to the size of an sms. It then sends these message parts as individual messages, each containing the sender's phone number. If the telecommunication network and operators are reliable, the messages will be delivered to the right phone. Otherwise, the message could be incomprehensible because the receiving sequence of messages does not match the sending one or worst, the recipient might not get any of them. In short, TCP provides a reliable but more resource intensive model of communication while UDP is lighter on the network but is not as reliable and the recipient application is the one responsible for ensuring that messages are received in the correct order. The Socket and ServerSocket objects introduced in Section 2.2 as well as the programming model of extracting I/O streams are employed to build connection-oriented applications which favour the strategy of continuous flow of data. However, UDP type communication is achieved through DatagramSocket objects which can be used to send and receive data and, DatagramPacket objects (or shortly, datagrams) which play the role of the sms messages in the above example. The programming model here is also based on the open-read-write-close concept. To this end, both the sender and receiver applications create a DatagramSocket on a particular port at their respec-

tive computers. Then the sender converts its message into an array of bytes and use it together with port number and IP address of the listening DatagramSocket to create a datagram. It then sends the datagram through its DatagramSocket to the other one. The routers on the Internet will take in charge of the responsibility to guide the datagram to its destination. One programming constraint in UDP-based applications is that the size of the array has to be predetermined and remain constant. Hence, when a DatagramSocket receives a datagram, it will know the exact amount of bytes to read from it [Harold, 2000]. When the receiving DatagramSocket object reads data from the network, it stores it in a DatagramPacket object that can hold a maximum of $65,507$ bytes. Now, if the number of bytes to be written or read exceeds the predetermined capacity of datagram, the excess bytes will be lost. Therefore, the programming approach here is that the sending application uses at least $sizeof(data)/65,507$ datagrams to send a message and at the recipient's end, the datagrams are buffered upon arrival and ensured that they are all there and that the sending sequence is preserved. Only then, the message can be restored and passed to the waiting methods.

2.2.2 Secure Socket

So far we have supposed that, when data is written to a socket, it is transmitted as plain text. This is very dangerous when sending sensitive information such as, passwords. To elaborate, when data is conveyed over a network, it is in the form of signals which are freely available to anybody with unrestricted access to that network media. For example, consider a person trying to log onto a web site in a cybercafe. When the person sends the authorisation data by entering it at the web page, the data are converted to electrical signals by the network card in the sending computer and sent through the cables that make up the network medium in the cybercafe. If on that network there is a computer whose network card is in a "promiscuous mode"[4], then the latter will siphon these signals and the person using that computer can read the transmitted data. So, instead of sending plain text data, the message must be encrypted before hand. Thus, even if the signals are intercepted as mentioned above, the eavesdropper will only get it crypted. This encryption/decryption process at two communication end-points can be automated by using sockets equipped with cyphering mechanisms. Until very recently, these special sockets were only available through an extension that plugged into the JRE because the law in certain countries view cryptographic

[4] `http://www.microsoft.com/windows2000/en/server/help/cgloss.htm`

softwares as threats to their national security [Oaks, 2001]. This extension known as the Java Secure Socket Extension (JSSE) is nowadays a part of the core JRE and is available in the Java Development Kit (JDK) since the release of version 1.4 [Harold, 2000].

With respect to the OSI model, the protocol for data exchange between sockets is defined at the *session* layer. Currently, Java (version 1.4.2) provides support for this layer by providing implementations for two secure protocols endorsed by the Internet Engineering Task Force (IETF), version 3.0 of the Secure Sockets Layer (SSL) and version 1.0 of the Transport Layer Security (TLS) [Hunter and Crawford, 2000]. So, the SSL and TLS protocols are implemented by the SSLSocket object which inherits from the Socket and therefore means that secure connection is only provided for TCP based applications. From a programming point of view the approach is identical as with normal sockets except for how the secure sockets are created. These implementations are based on the Java Cryptographic Architecture (JCA) framework and therefore permits the independent usage of Cryptographic Service Providers (CSPs). A CSP is a collection of implementations of various cryptographic elements such as, digital signature algorithms, certificate factories, key management services, etc. [Sun, 2004b, Sun, 2002] The current version of Java comes with a default CSP called SunJSSE and it provides an adequate set of these cryptographic processes. This CSP has also built-in capabilities of using other CSPs from other extensions such as, the Java Cryptographic Extension (JCE) [Sun, 2004a].

2.2.3 Multicast Socket

Communication on the Internet can be carried out in two ways, either from point to point like we have already seen (also known as unicast) or from point to multipoint. In the former case, if a server wants to transmit data to N clients, then it writes the same data to the N different channels that connects it to each of the clients. The second form of communication exists in two modes, *broadcasting* and *multicasting*. In the case of broadcasting, a server writes the data once to the network and it propagates until it reaches all the clients even those that have not requested the data. Java's DatagramPacket and DatagramSocket can be used to implement broadcasting as a mode of communication. On the other hand, multicasting is more sophisticated in the sense that the data only reaches those clients on the network that have subscribed to receive information from that server [Deering, 1989]. Indeed. multicasting achieves the same results as multiple

unicasting but with $N-1$ times less effort. The first practical IP multicasting was carried out in 1992 through the Multitasking Internet Backbone (MBone) project and the routers were modifed Sun Sparc UNIX computers and unicast tunnelling was used as well. Since then, multicasting has been in constant re-invention and many industry leaders like CISCO have joined the effort to stabilize and standardise the concept [Intelligraphics, 2004]. Nowadays, multicasting is a collection of five protocols and there are specialized hardwares called *multicasting routers* or shortly, mrouters, that are used in the process [Oosthoek, 1997]. The Java language also provides support for multicasting in the form of UDP datagrams addressed to a multicasting group. The programming approach in Java, is that both sending and receiving parties create a MulticastSocket object and use it to join the multicast session they are interested in. It must be pointed out that the latter object is an extension of the previously encountered DatagramSocket and thus multicasting uses datagrams to communicate as well. When the application wishes to terminate gracefully, it uses the leaveGroup() method to unsubscribe from the session [Farley, 1998]. Furthermore, the Time To Live (TTL) field present in the datagram is used to control the reachability of the packet on the Internet. This value is decremented each time it goes through an mrouter and when it reads zero, it is discarded. It must be noted that multicast addressing in IP uses the class D IP addresses which is in the range of 224.0.0.0 and 239.255.255.255. However, it is not necessary for all participating computers to have an IP address in that range as the clients need to only register themselves with a multicasting group that share a multicasting IP address and all the data sent to that address will be distributed to them. The registration process informs the routers between the client and the server that they can forward to this client and it also indicates to the local host that it should accept datagrams addressed to the multicast group [Harold, 2000].

2.3 Remote Method Invocation (RMI)

Networking applications can be categorized into two groups, those that transfer data between two hosts and those where one of the hosts runs processes on the other. The former groups applications that use protocols such as FTP and SMTP, while the latter type concerns applications that employ protocols like, Telnet, Secure SHell (SSH) or Remote Procedure Call (RPC). RPC is a protocol developed by Sun Microsystems the creator of Java, and they have incorporated the concept in the language. Indeed, RPC is incarnated as RMI which are se-

mantically equivalent but RMI is more sophisticated and powerful [Harold, 2000]. One advantage of RMI over RPC is that in the latter case only primitive data types can be passed as arguments while with the former, whole objects can be passed as parameters to methods which can in turn have as return values, objects.

RMI allows an application to call a method in an object living in a different JRE. The motivation behind this infrastructure lies at the programming level, where it creates the illusion of invoking a remote method to appear just like calling a local one. RMI based applications are made up of *clients* that access remote objects located on a *server*. The term client generally refers to the application itself or more precisely implies the local object in the application that makes the request to the server. There is a third component to this architecture called the *rmiregistry* (or in short, *registry*) which acts as a naming service and as an object manager and locator. The server records in the registry a list of those resident objects that will be accessed from afar by clients. Consequently, the registry assigns to those objects unique name handles that the clients will use to refer to them. Hence, when a client makes a remote invocation, it first consults the registry with the handle it believes is associated to the intended object. If the handle matches an object, it returns a reference to that object and the client makes a proper casting of the reference to the correct data type. For example, `MyStudent s = (MyStudent) Naming.lookup("student");` shows how a reference is to an object of type `MyStudent` is obtained using the handle `"student"`. If this operation completes successfully, that is, no *RemoteException* has been thrown, the typecasted object can be used just like any local object. Indeed, RMI hides from the programmer details such as socket communications with the registry and the remote objects or the loading of bytecodes in the appropriate JVMs when remote references are passed or returned [Farley, 1998].

Just as in RPC, RMI uses an interplay of *stubs* and *skeletons* to facilitate the process of calling remote methods. In the RMI communication model summarised by Fig. 2.3, we note that, when the client invokes a remote method, it actually calls a local method from the stub that has an identical signature to the remote one. This is the first step towards the goal of transparency marketed by RMI, where the stub acts as the proxy of a corresponding object in the remote JVM. It is in fact the stub/skeleton interaction that embodies the client/server paradigm of RMI. Moreover, the stub and the skeleton communicate using TCP/IP and follows the socket programming method discussed above (see Sect. 2.2). When the skeleton has restored the data sent by the stub, it calls the appropriate method in the proper object lying in the above layer and passes to it the parameters

that have been sent. After the remote method has completed its execution, the skeleton captures any returned values and serializes them through its output stream connected to the waiting stub. The latter is subsequently notified and it reads the value returned by the method call as its input stream deserializes it. The stub then relays this information to the calling method in the client layer thereby completing the last step of the illusion [Sun, 2003].

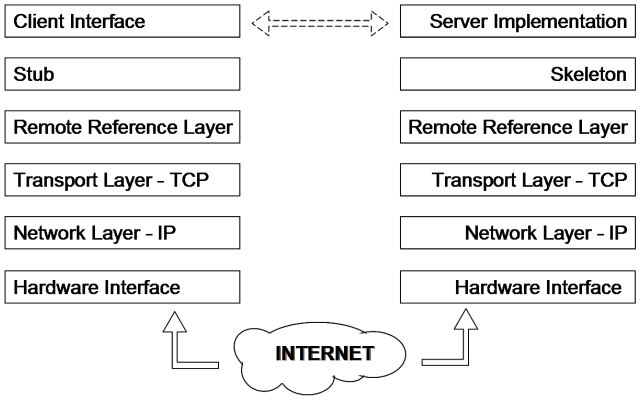

Figure 2.3 A layered representation based on the OSI model for the communication mode employed by RMI to invoke remote objects.

An interesting feature with RMI is the ability to carry out dynamic class loading just as in the case of applets, where the JVM downloads the necessary classes from a specified CODEBASE property in order to construct the applet on the local computer [Rodley, 1996]. As mentioned, RMI uses serialization through data streams to transfer information from one JRE to another and in the process, it also annotates the output stream with relevant information so that the appropriate class files can be loaded on the target JVM [Sun, 2003]. So for example, when a server registers an object in the registry, the CODEBASE for that object's class definition is recovered during the unmarshalling process and recorded with the reference to the remote object. Therefore, when a client requests a reference to the remote object and the stub cannot be found in its local CLASSPATH system property, it will download the class definition from the remote object's codebase and load it in its local JVM [Allamaraju *et al.*, 2001]. A major drawback with RMI based applications is the huge number of listening skeletons that can be generated and nowadays with firewalls protecting almost every single computer on the Internet, this is not a viable approach to create distributed applications. Fortunately, there are alternatives such as tunnelling RMI calls through HTTP

or using SOCKS protocol but these really degrade the performance of the applications [Sun, 2003]. Another important feature of RMI is *object activation*. Consider a distributed system where the new remote objects that are accessed remain alive in the JVM even after the clients are done with using it. After a while, the performance of the system would degrade because there would be too many objects to handle and also perhaps because many of these objects could be hogging resources. Henceforth, this became the motivation to introduce object activation in the RMI architecture to prevent these undesirable situations from arising. According to Allamaraju *et al.* [2001, pp. 35-97], *"object activation allows objects to be activated on an as-needed basis"*. In other terms, remote objects on a server are made to be of type *activable* instead of the old *unicast server* type so that, when the objects are accessed for the first time via a method invocation, only then they are loaded into the JVM. The main disadvantage with object activation is the huge amount of programming effort required to write even simple RMI based applications [Allamaraju *et al.*, 2001]. However, it allows backward compatibility whereby, the same clients that operated with unicast remote objects can now just plug and play with servers based on activable objects.

2.4 Java Message Service (JMS)

Message Oriented Middleware (MOM) is a proven infrastructure built around an asynchronous communication model that allows a wide variety of systems to interoperate through the production and consumption of messages. MOM offers a number of advantages that has made it the leading tool to build heterogenous distributed systems. For example, MOM services are language and platform independent and provide a number of communication models, point-to-point, push/pull and publish/subscribe that can be employed to solve different integration problems [Allamaraju *et al.*, 2001]. There are a number of excellent MOM products, such as IMB's MQSeries and Microsoft's Message Queue Server, but all of them are proprietary and commercial systems. However quite recently, Sun Microsystems released its MOM equivalent in the form of the JMS as part of the Java 2 Enterprise Edition (J2EE) computing platform. JMS specifies how clients can interface with message servers in a standard fashion and also it describes interfaces that message providers can implement to offer services to clients [Hapner *et al.*, 2002].

A JMS application is made up of a few *providers* that have implemented the JMS specification for a particular messaging model and a number of *clients*, which

can be JMS-based or legacy ones, that can send and receive *messages*. Also, an application can contain a number of *administered objects* that are configured to allow the clients to become portable as they interact with different JMS providers. These objects are normally accessed via a Java Naming and Directory Interface (JNDI) service. A queue in the point-to-point communication model is an example of an administered object. There are a many different ways of implementing a JMS provider but the most common approach is that of a star network with a *message broker* at the center and satellite clients surrounding it. Unlike certain MOM products, the JMS specification proposes only two most common forms of messaging,

Publish and Subscribe (Pub/sub): This is a *one-to-many* model where a client publishes a message to a *topic* and the interested clients can subscribe to that topic if they want to be informed. A topic is a well-known node in a hierarchical structure that serves to organise an information base. Any newsgroup or a website like news.google.com are examples of topics where internauts can sign themselves to be informed about particular subjects.

Point-to-Point (PTP): In this mechanism, a client (*producer*) sends a message to another client (*consumer*) via a specific queue in much the same way as an email system. Every client has a persistent queue that sequences incoming messages and which the client can consult at any time. The JMS PTP model defines how a client can locate its queue, send and receive messages to and from it.

The messages that clients exchange are made up of a header which contains fields for routing and identification purposes, and a body which holds the actual payload being delivered. The JSM message model provides a unifying API that solves the main problem of compatibility between different systems that employ proprietary schemes for defining the header part of messages. It also provides support for conveying Extensible Markup Language (XML) data as well as Java objects in the body part of a message [Hapner *et al.*, 2002].

In both of the above messaging models, a client creates a *connection* to a message broker using the *Connection Factory* administered object from the JNDI. The client can then generate from this connection as many sessions (topic or queue based) it needs, each with their own properties (transactional, acknowledgeable). The acknowledgement property is designed for these situations when a client or a broker would like to be informed that a message has been successfully delivered.

[Allamaraju *et al.*, 2001] compares this technique to the handshaking mechanism present in TCP communication protocol. Next, the application extracts from the session object a message producer and a message consumer object for the purpose of bi-directional communication with the broker. In the case of the PTP model, these objects are respectively known as the QueueSender and QueueReceiver while in the Pub/Sub case, they are known as TopicPublisher and TopicSubscriber respectively. When a message is published, the client can specify with it three attributes that adds more character to the message. With the first characteristic, the message can be made non-persistent or persistent. In the latter case, it is logged in a stable storage so that if there is a system or power failure, the message can be recovered by the JMS provider. Secondly, a message can have a priority on a scale of one to ten which causes the message intended to a client to be delivered ahead of others of lower priority waiting in its queue. Finally, a message can have a durability determined by the TTL attribute and is used in the context of a *durable subscription*. The idea is that if a broker receives a message with a TTL of specified amount of milliseconds, it keeps that message for that time in hope that, during that time, all the *durable* subscribers would accept the message and acknowledge it. A durable subscriber differs from normal ones in the sense that if it goes offline, messages that have been published to its broker meanwhile must be kept until it receives and acknowledges them or, they expire. Finally, in the Pub/Sub model, the subscription process is by default synchronous. That is, when a subscriber invokes the receive() method to fetch messages from the broker, it blocks and can wait indefinitely until a publisher has produced a message on that topic it is interested in. The waiting time can be limited by using methods such as receive(long timeOut) or receiveNoWait() which returns null if there are no available messages. However, there is an asychronous technique based on message listeners that are attached to topics of interests during the subscription process. So that, when a message becomes available, the *registered* subscribers are alerted and they can retrieve it. It must be noted that this technique can be coupled with *message selectors* which allow a subscriber to be informed only about specific messages. But this strategy works best with text based messages which can be syntactically parsed by the selectors. In applications based on the PTP messaging model, the producer session objects send messages to the broker which channels them to the respective queues after scanning the headers. At the other end, the clients that own these queue create receiver session objects to access the incoming messages. These objects can do two things: they can simply dequeue the messages and process them or they can just browse through all the

messages and leave the queue intact. It can be easily understood that in this model, messages have indefinite durability and, by default are made persistent until they are consumed by the intented clients. The message receiving process is by default blocking and there are similar methods as in the Pub/Sub model that can be used to impose timeouts on the waiting period. Furthermore, message listeners can also be employed to make the operation asynchronous and hence free the main thread to carry out some other task.

An interesting feature in the JMS specification is that a message can be made into a transactional element by transmitting the message through a session that has been enabled to do so. According to Allamaraju *et al.* [2001], this is helpful in situations where large documents need to be transmitted as a set of JMS messages. A transascted session is made up of a series of transactions which are essentially small groupings of messages considered as atomic units of work [Hapner *et al.*, 2002]. When a producer sends the member messages of a transaction to a broker, the latter acknowledges each receipt and at the end, the producer application can commit the session which causes the broker to dispatch these recently received messages. If the producer decides to rollback, then the broker disposes of these messages. From the point of view of the consumer, when it commits, the broker flushes the messages away and when the consumer rollbacks, the broker resends the messages from the last unacknowlegded message [Hapner *et al.*, 2002, Allamaraju *et al.*, 2001]. Another useful feature of JMS is the *request/reply* mechanism that can make JMS based applications behave like RMI ones [Allamaraju *et al.*, 2001]. The JMS message model provides a header field JMSReplyTo which holds the destination client to which a reply to this message is warranted. This way, a JMS client can behave just like an RMI client when it requests the consumer of the message to perform some actions on its behalf and return a response message back to it. JMS provides basic facilities on which many forms of request/reply paradigms can be built such as one message request causes one message response or one message request causes yields many responses. The main lacking of JMS is that it is only a specification and therefore every module have to be built using the interfaces provided by the API. Furthermore, JMS does not specify how clients and providers could coorperate to resolve problems like load balancing or fault tolerance. Moreover, there is nothing in the API that can be used for preserving privacy and integrity of messages or describe how digital signatures or keys should be distributed to the clients. These are left at the specific implementation of the providers.

3 Problem Definition and Analysis

In order to set the guidelines for the research behind this thesis, the following definition is provided:

Collaboration is a collective effort whereby any number of participants from any geographical locations can communicate ideas and decisions in real time. Furthermore, the collaborative environment should be receptive to a wide variety of devices and must allow for the realtime manipulation of the shared data or resources.

The Internet is a network that spans the whole planet. Furthermore, it is accessible to everybody via a number of medium such as, electric cables, radio waves, etc, and through various devices ranging from desktop computers to mobile phones. Thus, the Internet proves to be the ideal terrain for building collaborative tools as it satisfies the condition of *reaching an unlimited number of participants*. It also meets the requirement of *connecting people from any geographical locations*. In Section 2.1, we learnt how a Java-based application can run on many operating systems and therefore on a wide variety of devices. So Java helps in satisfying the other criteria in our above definition, that is, *collaborative applications must not discriminate against devices*. However, it must be stressed that though there is a wide variety of JVMs for various types of devices, their similarity is defined by a very small set of common core libraries. Furthermore, the devices can be distinguished in terms of processing power and available memory. So this calls for double caution when developing Java applications for a wide target group of devices. For the purpose of this thesis, we will focus on applications that can be developed using the Java 2 Standard Edition (J2SE) and at most the Personal Java Edition (PE). Finally, in order to *allow for the real-time manipulation of shared data and resources*, our framework and CTK provide concurrency control and load balancing mechanisms. Our particular approach to the latter, requires that the framework also implements a synchronization strategy.

3.1 Classifying Collaborative Applications

One can differentiate between collaborative applications in other ways than being synchronous or asynchronous by using other criteria such as, the amount of network traffic generated. As a matter of fact, in order to assess the efficiency of the framework and the toolkit, and also to measure the quality of the collaborative applications that can be created with the latter and handled by the former, we

need more than one dimension. To this end, a wide spectrum of collaborative and distributed applications were observed and it was decided to use the following criteria:

Data Repository: This criterion describes how the data component of such systems is represented. There are two possibilities, centralized and distributed. An example of the former case would be reserving movie tickets on the WWW. Typically, a customer would visit the movie theatre's website and enter details about herself/himself, the movie she/he wants to watch and a suitable time at which the show is screened. These information are collected by a web application behind the movie reservation webpage, and are recorded in a database attached to the webserver. This is system is considered centralized because if we consider the actions of several customers with respect to the webserver, these are directed towards one single point, namely, the booking database. On the other hand, in the distributed case, this trait of the convergence of information is not enforced. Typically, the actions of some users are directed to some points on the system while the same actions by other users converge to several other points. The concept can be explained as a logical grouping of users into several centralized subsystems maybe based on the type of action they perform or their geographical locations. These subsystems coorperate according to some particular synchronization algorithm so that the collaborative system can proceed without violating the overall data part of the latter. The distributed scheme can be categorized into, full or partial replication, based on how the data components exist at the local sites in regards to the overall data part of the collaborative system. As it can be observed in Fig. 3.1, in the case of full replication, data at the local sites are identical while in the partial case, they are different and some sites can even have no data. Applications that implement the former subtype of data management would be those that deal with small data sets. For example, instant messengers like ICQ, where the conversations among a group of users are recorded at each participating computer. Partial replication, is best suited for those applications where the data is distributed to as to offload processing stress on one particular computer. Another reason could be in the nature of the data itself. An example of such a system would be a collaborative editor of UML diagrams for a large software project. For example, UML diagrams related to the graphical user interface could be on computer A while those

diagrams pertaining to database access can be on computer B operated by another team of developers. Yet, all the diagrams can be accessed by a top-level project manager.

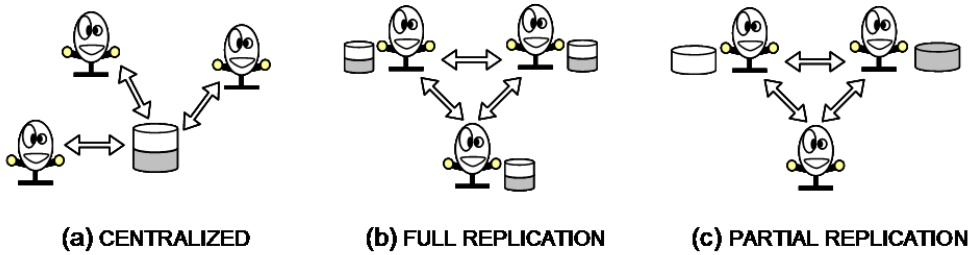

Figure 3.1 Three forms of data management approaches that can exist or co-exist in collaborative systems.

Collaboration Speed: Applications like 3D action games which under multi-player mode generate heavy network traffics and thus slow down the collaborative session. Heavy traffic can also cause the involved computers to slow down because they have to process large volumes of incoming data from their peers and prepare data to broadcast to them. Collaboration speed is an important factor to consider when designing collaborative applications. For the purpose of this thesis, collaborative speed is defined to be the sum of the duration of travel of a message and the time taken to process it when it reaches its destination. The unit of measurement is message per second or $msgs^{-1}$.

Concurrency Control: This criterion is used to distinguish between those applications that need some kind of data access management mechanism and those where it is not necessary. Examples, of the former type of applications are banking applications, where the execution of the users' actions has to be transactional. That is, it must be guaranteed that either all the tasks that make up the action are executed or none of them, the actions cannot violate the integrity of the data and all the actions must appear isolated from all other actions. To put these into context, consider the following three banking processes. First, a customer is transfering money from one of her/his account to another, the process cannot debit one account and not credit the other. That is, either both happen or neither. For the second transactional requirement, a customer cannot withdraw more money than

what her/his savings account hold. Lastly, suppose a customer is withdrawing money at a cashpoint and another banking process is calculating the interest gain on that account. The latter must do so either before or after the former operation but not in the middle of it. On the other hand, file sharing via peer-to-peer network is an example of applications that do not need such guarding mechanisms. If, for example, a user is downloading a file F from sites $\{S_1, S_2, S_3\}$ and the owner of F at S_1 decides to delete the file, it can do so even if the user is reading it at the same time. Unlike the former type of application, here the collaborative system will not fail and the user will subsequently carry on downloading F from $\{S_2, S_3\}$.

3.2 Qualitative Analysis of Network Topologies

In the domain of networking, devices in a Local Area Network (LAN) can be arranged in a number of ways in order to fit a particular physical or logical restriction. This particular arrangement of the involved hardware is known as the topology of the network. Fig. 3.2 illustrates five basic topologies which are used as such or collectively to create hybrid layouts. For example, the Internet is a hybrid of many topologies such as the *tree* when considering how DNS computers are interconnected or is a *star* topology with regards to web browsers and websites. As we will see shortly, each of these topologies comes with inherent structural and managerial advantages.

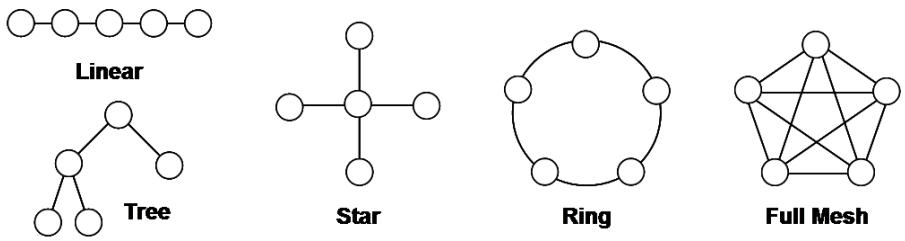

Figure 3.2 A graph representation of the basic configurations of computer (represented as nodes) interconnections in a network.

3.2.1 Structural Comparison

The *linear* layout is a chain-like structure where a member computer is connected to one or two other computer(s) depending on where it is located on the

chain. This setup is very fragile because if any of the computers goes offline then the chain is broken and therefore disrupts the collaborative session. In fact, the chance of such a thing happening is factored by $N-2$ where N is the number of computers in the chain. The *ring* approach is similar to the previous one except for the fact that the *first* and *last* nodes are connected to each other. This feature does not improve the fragility of the configuration but rather increases it to $N-1$. However, in this approach, if a connection fails, it reverts to a linear layout and collaboration can still proceed with slight alteration to the collaboration management mechanism. In both cases, the high risks of failure can be compensated through the use of a *next-door-computer reference* technique. If a computer loses connection to its immediate neighbour, it can then connect to the next neighbour since it has an IP reference to it and thus maintain the connectivity of the group. This approach is similar to that of the deletion or removal of a node in a linked list, where pointers are redirected so as to preserve the linear property of the data structure [Collins, 2005]. The *star* configuration is more robust compared to the previous one, as there is only one point of failure, namely the central computer referred to as the PoC (short for *P*oint *o*f *C*onvergence). This structural flaw can be remedied though the use of a backup computer that is in permanent synchronisation with the central computer while it is still operational. The *tree* arrangement is hierarchical and the topmost computer, known as the *root*, is the *parent* of the underneath computers that are attached to it and which under this circumstance are considered as *children* of the root. A child computer can itself be the parent of another computer and every child have exactly one parent. This layout is more fragile than the star topology but is more robust in comparison to either the linear or the ring configurations. In fact, for any child computer the risk of being disconnected from the group is factored by the number of computers joining it to the root, which is the *depth* of that node in the tree. This situation can also be helped by keeping an IP reference to the immediate grandparent. The last configuration is the *mesh* which can be *full* or *partial*. The difference is in the number of connections a computer can have to the others. In a full mesh arrangement of N computers, every computer has $N-1$ connections, each to one of the other computers, whereas in a partial mesh, the number of connections can be anything in the range of $[1, N-1]$. The higher the number of connections, the more robust the layout as it can withstand a couple of connections disruptions and still keep the computers connected to each other.

3.2.2 Managerial Comparison

With regards to the adopted selection criterion (see Sect. 3.1), speed of collaboration, is not affected by the choice of topologies. On the other hand, the latter can influence the design of a collaborative system with regards to the data storage management scheme and its performance in the context of the type of concurrency control mechanism employed. When comparing Fig. 3.1 and Fig. 3.2, one can notice the resemblance between the centralized and fully replicated systems, with the star and full mesh networks respectively. Indeed, already at the structural level these topologies provide a natural and straightforward way to implement these data management mechanism. Furthermore, the star network offers an intuitive and a simple approach for controlling concurrent events. Suppose that two or more collaborators send each a request to access the same piece of data. These messages travel to just one point in the star network. At the central point, there can be only one process that treats the messages in succession and replies to the collaborators in the same order. So, even though the accesses by the collaborators are concurrent, the requests are attended one by one in a specific order and thus does not violate the integrity of the data. The main drawback of the star system is that the central computer can quickly become overloaded if the rate of arrival of requests overwhelms that at which they are attended. As pointed out above, the full mesh structure is best suited for a distributed data repository implementation and in the case of partial replication, it reduces considerably the chances of system overload. However, the situation is different in the case of full data replication. More explicitly, a full mesh of N computers with full data replication can be considered as a collection of N star networks. This raises the issue of system overload which is typical to centralized systems. So, such systems are best suited for the situations where the involved data sets are smaller and there are not too much remote processing. Another concern is that of concurrency control. As explained earlier, the centralized nature of the star network instill a natural ordering on the incoming messages. But now, there are N star topologies operating concurrently and without an overall synchronisation. There is no guarantee that incoming events will be sequenced in the same order at all the sites. This is an essential requirement if the overall integrity of system is to be maintained. If the events are executed in different orders at different sites then the data repositories at these sites will be in different states and therefore violate the consistency of the system. So, with a full mesh approach, a sophisticated concurrency control mechanism is required as the topological structure

does not provide a natural one unlike the star configuration. The linear, ring and tree structural arrangements do not provide any advantages with respect to any of the collaborative characteristics introduced earlier (see Sect. 3.1). In fact, they tend to hamper the performance of the system by slowing the collaborative speed as discussed.

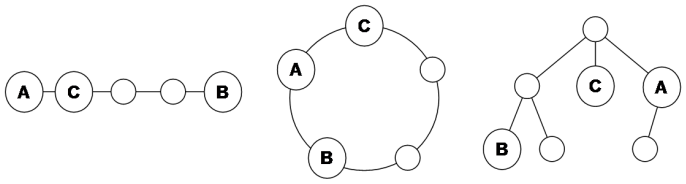

Figure 3.3 A, B are two arbitrary recipients of a message *broadcasted* onto the network by collaborator C.

For the sake of simplicity, assume that all the participating computers are alike and the network connections are uniform in terms of transfer rate. Furthermore, let the average collaboration speed between two neighbouring nodes on all three networks be $10msgs^{-1}$. We can observe that in all three cases (see Fig. 3.3), the collaboration speed between A and C is $10msgs^{-1}$ while that between B and C is $20msgs^{-1}$. However, in the cases of the star and the full mesh configurations, all collaborators would have a collaboration speed on $10msgs^{-1}$. Moreover, in the linear, ring and tree topologies, if for example A sends a request to B, then this would entail that all the intermediate nodes like C to relay the message until it reaches the target node. These nodes do some extra and unnecessary work by routing the messages to their right destination.

Based on the above discussions, it appears that the star and the full mesh topologies are the two best arrangements. However, the star network has an added advantage over the latter as its structure can be exploited to implement a very simple but robust concurrency control mechanism (see Sect. 3.3). Also, a full mesh approach would impose that all the end-point devices participating in the environment are powerful and resourceful enough to handle multiple connections and large amount of processing. Indeed, it was decided that the advantage provided by the star network outweighs the disadvantages of frequent system overload and a central point of failure. Also, the full mesh approach is not suited to our philosophy of collaborative systems as it discriminates against devices. For example, it is not reasonable for a small device with limited resources and processing power to hold multiple connections to several other devices. Therefore,

our adopted topology, as seen in Fig. 3.4, is a star network with a backup PoC. The latter remains in a sleep mode while the primary one is still functioning. The problem of system overload is discussed in Section. 3.4.

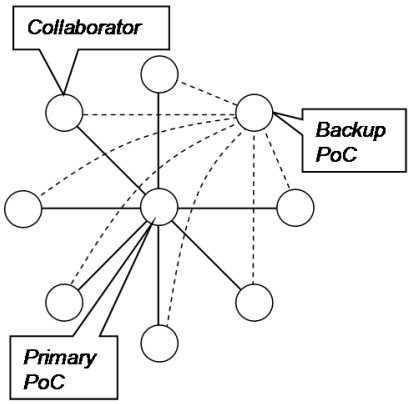

Figure 3.4 The adopted configuration for our collaborative model. The dashed lines represent passive connections between the collaborators and the backup PoC. The solid ones represent active connections.

3.3 Concurrency Control

In collaborative systems, several users can manipulate some shared data or resources simultaneously and therefore it is vital to ensure that conflicting operations do not execute at the same time. For example, consider the situation where a user A would like to apply the following operations $\{x + 10, x/5, x - 4\}$ on the shared object x which has an initial value of 5. Furthermore, assume that users B, C and D will each read the state of x in succession after A has applied one operation. So, after the first operation, B reads that x contains 15, then after the second operation, C reads that x holds 3, and finally after the third operation, D reads 1. So this situation leads to three different readings on the value of x as if the data was in an unstable state. This issue could become more problematic if, for example, any of B, C or D decided to use these intermediate values in some other operation or worst tried to modify x at the same time as A. Indeed, *reads* during *writes* or *writes* during *writes* are conflicting operations and should be avoided at all costs as they produce invalid results and leaves the data in an inconsistent state. To guard against such situations, a mechanism called *concurrency control* must be implemented that monitors these conflicting operations.

The simplest and most common approach is by using the principle mutual exclusion with respect to the critical regions in the data. For example, in the Java implementations of Objects, there is a module called a *monitor* which ensures that one and only one thread at a time can access the object (see Appendix A). In distributed systems, this approach is known as *locking* and the monitor equivalent is called a *scheduler*. When a process or a user is done with an object or a piece of data, the scheduler releases the lock and the latter can be contended by other waiting processes or requests. Chow and Johnson [1998] and Tanenbaum and van Steen [2002] propose an advanced form of locking mechanism called *two-phase locking* (2PL) which is based on the simple principle that, an executing process goes through two phases. In one phase it gathers all the locks it needs, and in the other phase it releases all the locks. For example, if user A above needed to perform some operations on a data object, say y, in addition to the above set, then in the first phase, A would try to acquire locks on x and y before it applies any of the operations. If it manages to get only one lock, it waits until a lock on the other object is granted to it by the scheduler. After it has applied all the operations, it releases the locks on x and y which will then be granted to other users, say B and C, who were waiting next in line, say for objects y and x, respectively. The main strength of 2PL is that is guarantees *serializability* at the expense of concurrency. More explicitly, users B or C have to wait for A to complete before they can apply their operations no matter how simple these operations are or how quickly they can execute their operations. Ideally, if user B need to execute only $\{y - 3\}$, then it could have been granted the lock to y, while A is busy operating on x, and way before A is done with x, B would have released the lock on y which could subsequently be acquired by A. In this case, we have two users operating simultaneously with non-conflicting operations. But as mentioned above, 2PL is a brutal approach that rather ensures at the cost of concurrency, that no operations can ever enter a conflict. Indeed, 2PL organises the sets of operations to be executed in a linear order much like a series which is the origin of the term serializability. One main problem with locking is that it can yield deadlock situations, which are highly undesirable. Assume that users A and B operate both on data objects x and y but in opposing order in an environment that has implemented 2PL for concurrency control. When the users start to execute their sets of operations, user A sees that it needs a lock on x and it is granted it. Similarly, user B is granted a lock on data object y. Since the principle requires that all locks must be collected, both users issue a request to the scheduler for the remaining locks. But as these objects are already held by the

other user, both enter a wait state which can last indefinitely and consequently creates a deadlock situation. Of course there are techniques to resolve and prevent deadlock but it complicates the process of concurrency control. However, 2PL is very useful for certain types of applications such as those based on transactions. However, it can be an overkill or even to expensive in terms of processing time in other applications and therefore does not qualify as a generic solution for the scope of this thesis. The proposed solution also sacrifices concurrency at the cost of guaranteed serializability but unlike 2PL, is deadlock free. The approach is simple and is centered around the topological design of our collaboration model (see Sect. 3.2). From Fig. 3.4, all the clients that need to apply an operation send a message to the PoC where they will be ordered in an arrival queue and treated one at a time. This way, all operations are made de facto serializable and their individual handling by the only execution process isolates them from potentially conflicting operations and also guarantees that no deadlock can ever occur. It is clear that this approach impedes considerably on the concurrent capabilities of collaborative applications built on this model. But, when taking into account the fact that this centralised approach also ensures a consistent distribution of identical sets of operations to all the collaborators. One has to agree that the penalty of lowered concurrency can be overlooked by considering these guarantees.

3.4 Performance Bottleneck

The structure of our collaborative network provides a robust foundation for implementing a concurrency control mechanism. However, this structural organisation also brings with it the nasty problem of performance bottleneck, as all traffic is focused on the PoC. To help relieve the problem, the PoC is fitted with a queue that buffers the incoming traffic and thus prevents the collaborators requests to be ignored or lost. But the problem still persists and it is now in terms of growing waiting periods before a client's request can take effect.

This problem can be tackled quite easily if we assume that all the collaborators on average produce the same volume of traffic and thus from the PoC's point of view all the collaborators are equivalent in this context. It must be stressed that the collaborators are still different if we consider their connection latency with respect to the PoC. This criterion is discussed in Section 4.1.1. So, given now that all the collaborators are potentially the same, the straightforward strategy is to create an extra PoC and instruct half of the collaborators to disconnect and reconnect to the latter. The traffic volume has not changed but, now each

PoC is only exposed to a fraction of it (half in this case). Fig. 3.5 illustrates this approach. An *island*, which is a grouping of the main PoC, a backup PoC and a number of collaborators, is a smaller version of initial network layout. The islands can operate independently from each other. However, they should not do so but instead coorperate so that the collaboration session is preserved. The strategy needed to make the various PoCs coorperate so that the collaborative session is not disrupted is addressed in the following section. This fragmentation approach into independent computing units has the advantage that it makes the collaborative system easily scalable. New islands can be formed elsewhere and join the session while old ones can leave whenever they want.

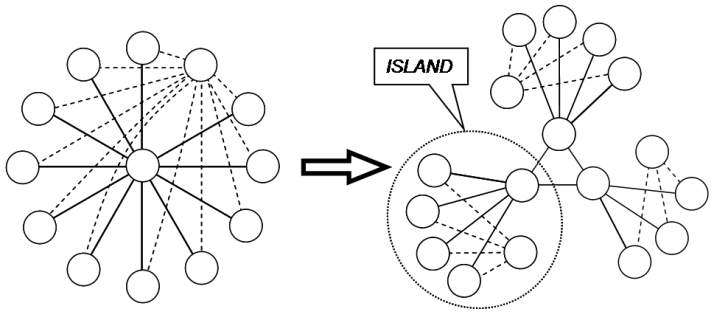

Figure 3.5 Solving the problem of performance bottleneck by distributing the connections load on several *islands* of computing.

3.5 Synchronisation

The adopted solution of creating islands of computing in order to resolve the problem of overload at a particular PoC invites an equally severe problem to our collaboration model. This problem, known as the *synchronisation problem*, can be understood by considering the following situation. Consider a configuration of two islands as illustrated in Fig. 3.6. Let $user_1$ and $user_4$ send update operations $op(x)_1$ and $op(x)_4$ at about the same time to modify the state of a common object x. When these operations reach their respective PoC, they are distributed to the other member users in the island and also to the peer PoC so that the other users in the adjacent islands can be informed. It is possible that due to a number of unpredictable events such as network lags, $user_2$ receives the operations in the following sequence $\{op(x)_1, op(x)_4\}$ and at $user_3$ it arrives in the reverse order. This can be still an acceptable situation provided that the operations are

commutative but if they are not, this would violate the integrity of the data. More explicitly, suppose that x holds an initial value of 15, and that $op(x)_1$ and $op(x)_4$ are actually $x - 2$ and $x + 5$ respectively. So, after $user_2$ applies the operations in their order arrival, the final result on x is 18 just like at $user_3$, because the operators + and − are mathematically commutative. However, if $op(x)_1$ was instead $x/2$, then $user_2$ would read the final state of x as 12.5, while at $user_3$, it would be 10. The operators / and + are not commutative. In this example, the operations are from the domain of mathematics and therefore sets of operations can be proven to be commutative. But in an arbritary domain where operations can be as abstract as, say *"broadcast smiley icons"*, it is painstaking to determine if all the operations in the context of the application would be commutative. As it can be deduced, using the principle of commutative operations to guide the creation of collaborative applications free of synchronisation problems is not a reasonable approach at all.

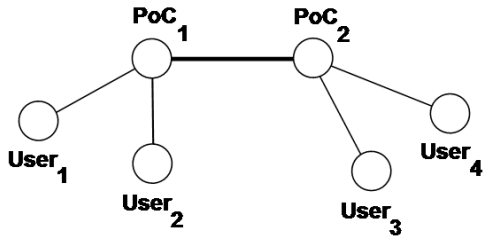

Figure 3.6 A typical setup of two islands connected to each other by their PoCs. PoC_1 distributes messages from users in its island to the users in the adjacent island via PoC_2.

However, as we have already seen, the star topology provides a simple mechanism for ordering and distributing operations and one can apply the same principle here to solve the synchronization problem. That is, one just has to add a top level node above all the PoCs. This makes it act as the PoC. In other terms, when a user sends an operation to its PoC, the latter forwards it to the top node which queues it with the other incoming operations from the other PoCs. At the other end of that queue, the earliest operations are popped out and distributed to all the PoCs which, in turn, send it to all of their respective satellite collaborators. As it was pointed out earlier, this approach guarantees the same order of distribution of operations to all the participating sites. However, this elegant solution is flawed with the same problem that made us fragment the otherwise robust star topology into islands, that is, performance bottleneck. Consequently,

we will look into a non-centralized solution to the problem of synchronisation. It is important to note that the above discussion only highlights the sychronization problem under a specific light of somehow queueing the, incoming operations at their destination. The other way to look at the synchronisation problem is that the operations are not queued at all because the users can process the operations very quickly or, the second operation took so long to arrive that the first one has been handled already. Under these circumstances, there is nothing that can be done to solve the problem of synchronisation.

Tanenbaum and van Steen [2002] write about a number of techniques that can be used to solve this problem, provided it is under the queued condition. The idea is to associate to the operations a countable property like a temporal value or a sequence number so that a total ordering using a *happens-before* relationship can be established on the queue. The simplest case appears if all the points to be synchronised are at the same time and their respective clocks tick at exactly the same speed. Then the messages could be assigned a timestamp and the incoming messages could thus be positioned at the right position in the queue. Since the time is universal and the ordering rule the same, the queues would at best be identical but otherwise subsets of each other as the consumption rate of queue items might differ. However, as pointed out by Tanenbaum and van Steen [2002], synchronisation techniques such as Cristian's or Berkeley's algorithms based on physical clocks are not very precise. This is because clocks are considered to be synchronised if they can still agree within a small margin of error. Therefore, it is impossible to distinguish between two events that occur after one another but within a time frame as small as the margin of error. Lamport [1978], proposes an alternative approach called *logical clocks*, to the synchronisation problem. It is100% accurate unlike the former approach. Moreover, it is intrinsic to the messaging model and eliminates the risks of failures that exists with synchronisation techniques based on physical clocks whereby external processes are needed to constantly monitor the synchronicity.

Our synchronisation mechanism is based on Lamport's timestamp technique and we still make use of a top-level node that connects all the PoCs. However, the activity of this node is considerably lower than the one proposed in the previous solution. The top-level node is called the *ticket server* (TS) and it is responsible for allocating fresh *bundles* of *tickets* to all the PoCs whenever they request it. A ticket is simply a positive integer, which makes a bundle, just range of integers. Our approach relies on the fact that when the TS creates a new bundle of tickets and broadcasts it, all the PoCs will receive identical copies of the tickets. As an

example, consider again the situation in conjunction with Fig. 3.6. Suppose $user_1$ sends operation $op(x)_1$ to PoC_1, the latter then retrieves the smallest available ticket from its bundle, say t_1, and it demands all the other PoCs to acknowledge that it can use it. When a PoC, say PoC_2, receives the request about the ticket, it verifies the value of the smallest available ticket in its own bundle. Assume that this ticket is t_2, and that it is greater than the one PoC_1 wants to use. So PoC_2 removes t_2 from its bundle and replies to PoC_1 with it. On the other hand, if t_2 is smaller than t_1, then PoC_2 removes all the comparatively older tickets from its bundle until the resulting smallest available ticket matches t_1 and it then acknowledges to PoC_1 with t_1. Note that at this point, the next smallest available ticket on PoC_2 is t_1+1. Now, when PoC_1 receives the acknowledgement messages from all the other PoCs, it verifies if any of them has a response ticket value greater than t_1. If this is the case, PoC_1 removes tickets from its bundle, until the smallest available one is equal to the one it found from the acknowledgements. PoC_1 then attaches that ticket to $op(x)_1$ and sends it to all the users in the island as well as to the other PoCs.

4 Design

We saw in Chapter 2, that RMI (see Sect. 2.3) and JMS (see Sect. 2.4) are two different approaches provided by Java that can be used to design and build collaborative applications. Though they are quite different in many respects, both are suitable to build synchronous as well as asynchronous applications. However, like all the other technologies from the Java family, their main drawback is their lack of support mechanisms such as concurrent data access management. Our design answers to this issue provide a concrete solution to the problem of distributing the workload of the system and consequently to the implied synchronisation problem. The toolkit proposed in this thesis is based on the socket programming strategy discussed in Section 2.2. A multicasting approach (see Sect. 2.2.3) would be best suited for implementing the communication modules of collaborative applications but multicasting is not yet widely supported on the Internet. Therefore, it will not be considered as a communication model. However, we simulate the multicasting approach through multiple unicasting and based the Pub/Sub model from JMS. Furthermore, we base our communication model on a principle of remote execution similar to RMI (see Sect. 2.3) but reflection (see Sect. 2.1) is employed to achieve the same effect. Even though this method is quite slow, we hope that with advancement in technologies, the involved delays would be shortened. On the other hand, this is a much simpler programming method than the RMI approach. Moreover, the design of our framework favours a connection-oriented style of communication (see Sect. 2.2.1) as it has inherent security features from TCP and Java offers secure sockets (see Sect. 2.2.2) based on this model.

The star network provides a simple, elegant and robust solution to the problem of concurrent access (see Sect. 3.3). Even though the presence of a central process (PoC), overseeing the collaborative session, guarantees the serializablity of these operations, the performance bottleneck problem arises. It was decided to resolve this problem by spliting the large star topology into mini star topologies and thus distribute the connection loads over many islands. This simple strategy inherits the main problem of sychronisation between the islands so that the collaborative session is consistent from a global point of view. This problem is tackled by the implementation of a ticketing service that helps to bring a sense of total ordering among queued operations waiting to be attended. This overall approach facilitates the scalability of the collaborative session as islands can join or leave the net of PoCs at will. In Section 4.3, we see how this dynamic topology can also be

used to implement a collaborative system with the sharing of *partially replicated* data as shown in Fig. 3.1*(c)*. But first of all, we describe our collaboration model in terms of the roles played by all the elements.

4.1 The Collaboration Model

Our collaboration model is made up of *coordinators* and *participants* which communicate with each other via the exchange of *capsules*, as it can be seen in Fig. 4.1. Our model also employs a full replication strategy when it comes to data management. This allows any of the participants to be considered as a coordinator in the future.

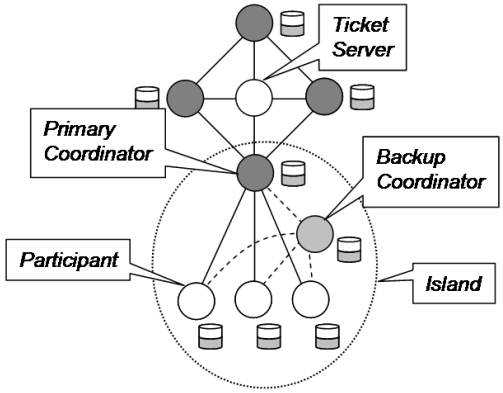

Figure 4.1 The adopted topology for the collaboration model is a collection of star networks with their central units connected to each other in a full mesh configuration.

4.1.1 Coordinator

As the name suggests, the coordinator oversees the smooth running of the collaborative system and in our case, it is a central processing unit as shown in the island in Fig. 4.1. The role of the coordinator is to force a linear order on concurrent requests made by the satellite participants. The coordinator possesses an up-to-date copy of the data found at all of its participants. So, whenever it receives an update request from one of its participant, it tries to apply it on its copy before authorising and distributing it. At a coordinator, all incoming requests are treated by only one process. This ensures a state of mutual exclusion when there are two or more requests that need to access the same piece of data.

Even though the model makes use of the one-access-at-a-time principle, it does employ parallelism so as to increase the efficiency of the system. As a matter of fact, a coordinator has four services that run simultaneously,

Main Service: Using the Main service the coordinator listens to connection requests from potential participants. They create a protected connection by opening a secure socket (see Sect. 2.2.2) to this service using the known (published) IP address and port number of the coordinator. The participant then sends a username/password pair identifying itself as well as the signature of the collaborative application to be used with this coordinator. If the collaborative application is intended for this environment and if the participant has been correctly identified, the coordinator then sends some connection data otherwise it closes its socket connection with that participant and the latter cannot proceed further. The connection data contains the port numbers of the other available services, *admin*, *data* and *view*, and the IP address and port number of the Main Service of the Backup coordinator in this island. These data are bundled into a self-executable capsule (see Sect. 4.1.3) which automates the connection process to these services at the participant. While these connections are being established, the main service freezes the other services and all incoming requests to the coordinator are queued. Once the new participant has established all the other services, it informs the Main Service which then brings up-to-date the local copy of the data at the new participant. This can be done, either by replaying a history of updates on the participant's data or by sending a serialized copy of the data to the participant. According to the session initiation protocol, if the updating process is successful, the participants sends an acknowledgement message. Otherwise, it requests to the Main Service to start over. When this service receives the *ok* message, it unfreezes the other services and collaboration can resume as before. The Main Service is equipped with a time-out mechanism that disconnects the participant if it is taking *too long* to respond. At the end of a new connection process, the Main Service goes back into listening mode for other potential participants.

Admin Service: The Admin service is mainly used to gather statistical data about the participants and it gives an idea about the "health" of the collaborative environment. The coordinator regularly requests from the participants to measure their *responsiveness* with respect to all or most of

the other participants. The responsiveness between participants a and b, measured in milliseconds, is calculated as,

$$R_a^b = T_a^b + P_b + T_b^a + P_a$$

where T_x^y is the time taken for a capsule to travel from x to y and P_x is the time taken for x to process the capsule. The coordinator also requests from the participants to send the amount of free memory footprint available in the JRE. These data are intended to enhance the collaborative experience by improving the responsiveness of the participants by finding optimal configurations. As an example, consider the situation illustrated in Fig. 4.2*(a)*, where there are six participants from Finland and Sweden in an island connected to a coordinator located on the same computer as Swe_2.

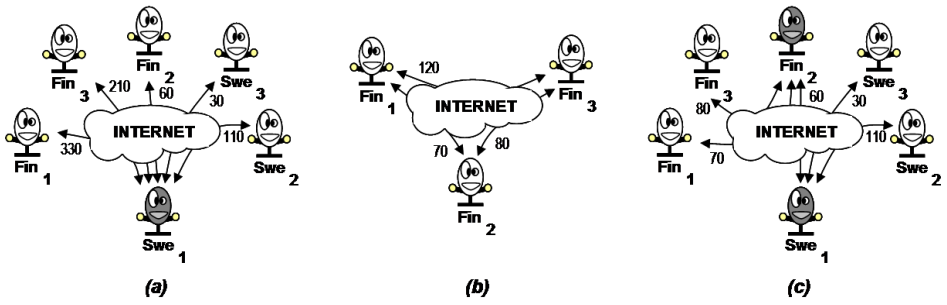

Figure 4.2 *(a)* The responsiveness of all participants with respect to Swe_2. *(b)* The responsiveness among a sample of the participants. (c) New responsiveness after a *network redraw*.

Using this service, the coordinator sends a request capsule to all the participants to measure their responsiveness with respect to each other. The capsule contains a list of the IPs and port numbers of the UDP listening sockets (see Sect. 2.2.1) for every participants in the island. These sockets are created during the connection phase when the participants join the island. So, with this capsule, all the participants in the island know the listening address of their peers and can thus send a *measurement capsule* to them to calculate their relative responsiveness. This capsule contains three fields, the time at which this capsule has been created, the IP address of the sender and the Port number to which this capsule must be returned when the recipient does so. The capsule also contains the necessary instructions

that will automate this replying process when it reaches the recipient's site. Fig. 4.2*(b)* shows a sample of responsiveness among a specific group of participants from Finland. Although the given figures are fictional, it is easy to agree that the responsiveness gets smaller when considering a smaller geographic locations (for example, just Finland or, a building or, a room) than a larger one (say, both Finland and Sweden). When the participants send these collected data to the coordinator, they are analysed by a special algorithm which determines how to reduce the responsiveness by introducing new coordinators in the network. For example, Fin_2 seems to be a good candidate and the coordinator at Swe_2 triggers a new coordinator at Fin_2 and instructs Fin_1 and Fin_3 to disconnect from Swe_2 and connect to F_2. Also, Fin_2 and Swe_2 establish a bidirectional connection between them and connect to a Ticket server which is started on either of them depending on which one has more free memory. This whole process in known as a *network redraw* because the topology of the collaboration environment is changed as it can be noted in Fig. 4.2*(a)* and *(c)*. At the moment, network redraws are not implemented to improve the responsiveness of the environment. Statistical data are still collected for the study of the behaviour of collaborative systems in terms of network traffic, types of operations and their frequency. However, network redraws are still performed but to balance the workload of the network by reducing the ratio of participants to coordinators, that is, to create more islands. At the moment, the memory footprint information from the participants are used to elect backup coordinators. This is discussed below.

Data Service: Data service accepts capsules that contain operations that require exclusive access. Typically, they are those that perform reads and writes on the state of the data and this service inherits its name because of this reason. Just like the Admin Service, this one also has a capsule analyser (see Sect. 4.2.4) attached to it. Whenever a participant requires to update the state of the data it first requests the coordinator to validate the operation by applying it on its own copy. Since the coordinator is at the point of convergence of all messages and all operations are executed under a mutually exclusive climate, it is easy to agree that the local copy of the data is fully consistent. If the requested operation executes without any problem, the coordinator authorizes the participant to carry on and distribute the capsule to all participants so that they too maintain a consistent copy

of the data.

There are two coordinators in every island, the Primary and the Backup. The Backup coordinator always remain in a sleep mode while the Primary one is operational. Whenever the Data Service at the Primary coordinator broadcasts a capsule to all the participants, it always sends to the Backup coordinator as well, so that it maintains a consistent copy of the data. This is necessary for the situation when the Primary coordinator becomes offline and the Backup coordinator has to take on the role of the Primary coordinator. At this point, the local repository must be in the latest state so as to correlate with those at the participants' site. The Backup coordinator assumes this new role when the first participant connects to it. The newly formed Primary coordinator also delegates the role of Backup coordinator immediately to that participant instantly so as to reduce the window during which the collaborative system can collapse due to two close successive failures of the Primary coordinator. However, this participant might not be suitable as it might not have enough memory to behave as Primary coordinator if it is called upon to do so. But as mentioned above, the Admin Service regularly gathers memory footprint data from the participants. When a participant with a larger (this is a relative concept and can be set in the profile of the application) free memory space joins the island, the Primary coordinator starts a Backup coordinator in that participant's JRE. It also instructs the other Backup coordinator to shutdown after sending the address of the new Backup coordinator the other participants. Furthermore, this election process also takes place when the participant hosting a Backup coordinator goes offline.

View Service: View service handles those operations that do not require special attention. It gets its name from the Model-View-Controller design pattern where the View part deals only with operations that updates the viewport and not really the data (Model). Indeed, given a collaborative system such as a mindmap editor where some users are dragging nodes on the viewport around for a better layout, these actions are not necessarily affecting the state of the mindmap. This service is optional and thus those participants that want to be informed about for example, the presence of other users, must register with this service. Essentially, the View Service takes on the role of multicasting (see Sect. 2.2.3) as it distributes to a list of subscribers events that they wish to receive.

When it comes to the distribution of data between islands, the Data Service differs from the View Service in the sense that its needs to get acknowledgements (see Sect. 3.5) from all other coordinators before it can broadcast a request to its participants or other islands.

4.1.2 Participant

The participant is a generic term that groups a collaborative application which can be human driven or automated. An example of a human driven application is a mindmap editor and actually any application that requires human intervention via a user interface would fall in this category. On the other hand, an example of an automated collaborative application is a program that tries to discover the secret key that has been used to encrypt a message. A participant is always connected to the coordinator via three Channels, each connected to a service at the Primary coordinator. As mentioned earlier, when a participant joins a collaborative session, it first connects to the Primary coordinator of the island it wants to belong to. This address, which is a combination of an IP address and a port number, is public. After the connection is established, the participant sends a username and a password which is used for authentication and as an ID. It also sends a signature of the application that is used in the collaborative environment. This is needed because our framework can host a number of collaborative applications and therefore the system, rather the coordinator, must be able to discriminate against applications that are not suited for its group. From the point of view of the user, this connection process takes some time as the application needs to connect to the various services and also to update its local copy of the data. The participant also receives the address of Main Service of the Backup coordinator of the island. It can happen that the Primary coordinator dies for no apparent reasons and if the participant sends a request to the latter, it will be notified that the coordinator is not there anymore. The participant then contacts the Backup coordinator which at that point becomes the Primary coordinator of the island. In this case, connecting to the new Primary coordinator is less slower as it only involves connecting to the services and perhaps downloading a very short history of those operations that have occured from the point the new Primary coordinator was operational to the time that the participant has connected to it. Finally, no other participants can join this collaborative session as they are the only ones aware of the address of the now inactive Primary coordinator.

4.1.3 Capsules

Participants and coordinators communicate among each other through the exchange of capsules. Sometimes, these capsules are intended to be sent to everybody and other times, they are meant just for one particular target. In order to distinguish between these two forms of capsule distribution, the capsule holds a *Propagation field* which indicates how the capsule should be handled after processing its content. The capsule has also a payload of *Command Objects* which describes what the source party would like to execute on or communicate to the target one. Essentially, a Command Object is a delimited string which states the object and method to access at the destination. Table. 4.1 gives a formal definition of the payload where <Obj Name> and <Method Name> are valid class names and method names, respectively, according to Java's Language Specification.

<Payload>	:=	<Cmd Obj> | <Cmd Obj>$_{Sep_1}$<Cmd Obj>
<Cmd Obj>	:=	<Obj Name>$_{Sep_2}$<Method Name> |
		<Obj Name>$_{Sep_2}$<Method Name>$_{Sep_3}$<Args>
<Args>	:=	java.lang.String | <Args>$_{Sep_3}$<Args>
Sep_1	:=	#@#
Sep_2	:=	#+#
Sep_3	:=	#%#

Table 4.1: The syntactic structure of a capsule's payload.

The string ctk.core.participant#+#TX_connectToAdminService#%#4550 is an example of a Command Object (<Cmd Obj>) sent by a coordinator to force the recipient participant of this capsule to connect to its Admin Service. The object's name is referenced by its fully qualified name. This example also demonstrates that all parameters (and return values) are passed as string irrespective of their natural type. So, the last argument in the sample Command Object, the port number, is passed as a string even though semantically it is an integer. As a matter of fact, the method TX_connectToAdminService will be responsible to convert the parameters passed to it to their correct type before processing them. Moreover, the capsule design allows complicated actions to be created by grouping smaller ones through the concatenation of Command Objects. For example, the connection initiation capsule sent by the Primary coordinator actually con-

tains Command Objects to force the participant to connect to the Data and View Services as well. Furthermore, the payload is also made up of a Command Object that causes the creation of the UDP listening socket mentioned earlier and which is used to measure peer responsiveness. The design of the capsule also allows for an atomic execution capability similar to transactional operations [Coulouris *et al.*, 2001]. More explicitly, if we take into consideration the connection initiation capsule mentioned above, it possible to abort the whole connection process if the execution of any of the Command Objects fails. In other terms, if for example the participant cannot create a Data Channel to the Primary coordinator, then there is no point to create a View Channel or the UDP listening socket for measuring peer responsiveness. To this end, a capsule is considered *transactional* if at least one of the Command Objects in its payload is marked as being transactional. A transactional Command Object is recognised by the TX_ prefix to the method name and every such methods has a complement recognised by the prefix RE_ which is meant to restore the state of the data modified by the former method. So, the complement of the above method (TX_connectToAdminService), is RE_connectToAdminService and it is meant to tear down the Admin Channel when it is invoked. These are recommendations and the implementations of these methods are at the discretion of the collaborative application developer. This atomic execution property is ensured by the capsule analyser (see Sect. 4.2.4) which is responsible for automatically invoking the RE_ methods of the successfully executed previous TX_ methods in the current capsule.

4.2 The Software Architecture

The coordinators and participants need to perform a number of different parallel tasks. Consequently, the toolkit provides the concept of processes which are essentially empty shells that can be filled with code in order to perform the desired functions. The following section introduces CTK Processes and also describes how coordinators and participants use them to create components such as Services and Channels. In the final part of this section, we explain the operation of the execution engine of our framework.

4.2.1 CTK Processes

A CTK Process (or Process) serves the same purpose as a thread but it has more features. It can be safely stopped and restarted and it also has safety mechanisms

that allow it to recover from failures. In this thesis, a process can have at most one *parent* process while the latter can have zero or more *child* processes. An intuitive way of designing collaborative applications is to start from the top and progress to the bottom by identifying the main services that the application will provide and decomposing those into smaller ones which perform specific tasks. For example, in most collaborative applications, the communication module is made up of two processes, one that reads data from the network, and one that writes to it. Moreover, the reading process can also be devided into a process that strictly reads bytes from the network and another that interpretes them. In addition, the data interpreter process is dependent on the network reader one. This type of design exercise allows the creation dependency trees of related processes as illustrated in Fig. 4.3.

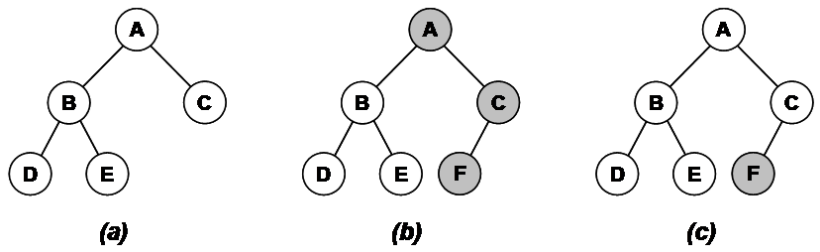

Figure 4.3 *(a)* A normal dependency tree of processes. *(b)* The contamination effect when critical process F is introduced. *(c)* A reverse contamination effect when the importance of process C is made non-critical.

This feature of parent/child relationship in CTK processes eases the development effort by making it simple to transit from the design to the implementation phase. The dependency tree serve as a road map of what processes are required and also clearly indicate how they relate to each other.

At the mechanical level, a process has as its core a thread (T_c) that carries out all the work when it is running. It is implemented as an abstract class that implements the java.lang.Runnable interface which creates an association between T_c and the abstract run() method from the interface. Thus, when the start() method of T_c is invoked, it is the run() method of the process that is called into action instead of that of T_c. The responsibility of the programmer is to fill the run() method with the correct statements so that the process can behave appropriately. A process can be in four states as illustrated in Fig. 4.4.

The *New Process* state is entered whenever T_c is given the reference to a

new thread object which can be in the constructor of the process object or the restart() method. This state is similar to the *New Thread* state from Fig. 6.1. The *Runnable* and *Paused* states are also equivalent to the *Runnable* and *Not Runnable* states from Fig. 6.1. However, unlike the *Dead* state from the latter diagram, the *Stopped* state in Fig. 4.4 is not terminal. The process can be restarted via the *restart()* methods which allows the thread to enter the *New Process* state again. It must be pointed out that a process can be started an unlimited number of times or a bound can be imposed by setting the attribute restart_limit to the desired number of times.

Every process is subject to hazardous terminations caused by unpredictable runtime exceptions and in some situations, it is important that a few of these processes operate all the time and thus become critically important to the application. If any of these processes die inadvertently, then the resulting situation can crash the whole application or render it useless. For example, if the saving feature of a text editor does not work, then there is no point to write documents with it. As a consequence, it has been decided to use a positive integer variable critical_value in the architecture of CTK Processes to identify the critical ones. A non-zero value indicates a critical process and if it is equal to zero the process is considered non-critical. When a critical process is added to a dependency tree, as in Fig. 4.3*(b)*, the critical_value of its parent is incremented by one, which in turn causes that of its parent to be incremented by the same amount and so forth. The same principle applies when the critical_value of a process is set to zero as illustrated in Fig. 4.3*(c)*. If a parent has two critical children, and one of them is made non-critical, then the parent remains critical because its critical_value is

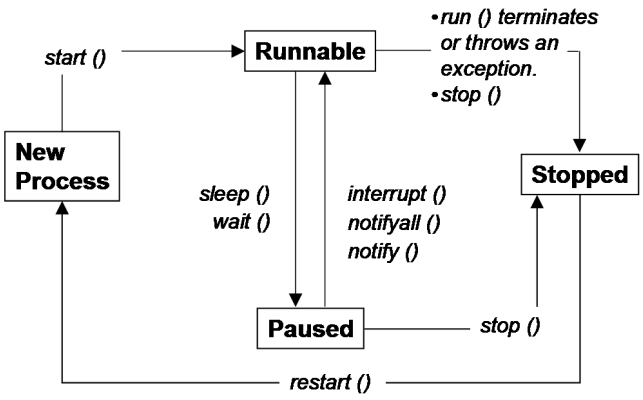

Figure 4.4 A state chart of the life cycle of a CTK Process.

decremented by one but is not yet zero. Furthermore, the avalanche effect of changing the critical_value always moves up the tree from where it was started. This has been deliberately made so, as it gives the possibility to create *optionally critical sub-trees* in an application. This is helpful for example in situations where at runtime it is collectively decided to sacrifice a particular feature of the application that is no longer working so that collaboration can still proceed. The motivation behind this is explained next.

The CTK process architecture provides an automated mechanism that allows critically important processes to be automatically restarted if the current number of restarts has not exceeded the restart_limit. This responsibility is given to a special process called the *Babysitter*. When a critical process is created, it is registered with a babysitter whose main role is to monitor its liveness during the running of an application. If a critical process suddenly becomes non-critical for some reason, then it is automatically removed from the watch of the Babysitter. The latter is just like any other CTK processes and is therefore also liable to hazards with terminal effects. To guard against this situation, the Babysitter has an unlimited number of restarts. It also monitors a special process called the *Watchdog* whose role is to watch over the liveness of the Babysitter and it too has an unlimited number of possible restarts. Indeed, this *Babysitter/Watchdog* mechanism provides a good technique of permanent monitoring of critical child processes. This mechanism is not completely foolproof as in the event that if both processes die in very close succession then the perpetuality property is broken. Now, if a critical process has exceeded its number of allowed restarts, then the Babysitter destroys the process which in turn causes its parent to destroy itself. This parent destruction effect propagate up the dependency tree and the application is eventually shut down if the root of the tree is reached. This catastrophic effect can be minimized as in the situations of optionally critical processes by overloading the destroy() method of the critical process to reset the critical_value of its parent. This will limit the destruction to the subtree with the critical process as the root.

4.2.2 Services

An example of the application of Processes in our framework is in the creation of the coordinator's Services. Fig. 4.5 illustrates the architecture of the Data Service. The same architecture is also adopted by the Main and Admin Services. The *Connection Listener* is another example of an implementation of a Process

and its role is to be on the permanent lookout for new connection requests from interested participants. So, when a Service receives such a request, the Connection Listener extracts from this connection a *connector* object and adds it to a list of currently active connectors. These are essentially server-side sockets to which the various participants are connected to when it is said that they are connected to the Service. The connector object provides to the Service two data streams: an input stream for reading from the network or the Internet and an output one to write to the same network. Moreover, these connectors are also Process implementations and their role is to constantly listen for incoming capsules from their respective participants. So, when a connector reads a capsule sent by its attached participant, it queues the former to a buffer similar to the *Incoming Capsules Buffer (ICB)* in Fig. 4.5. At the other end of the ICB, is the *Capsule Reader (CR)* whose role is to dequeue the ICB and pass the removed capsule to the *Capsule analyser (CA)* (see Sect. 4.2.4). After that it enters a waiting state. If there are no capsules in the ICB, the CR enters a sleeping state and is waken up whenever one of the connectors add a capsule to the ICB. After the CA has completed processing a capsule, it wakes up the CR which repeats the same exercise again by dequeueing another capsule from the ICB and passing it to the CA. However, if the CA has a capsule to return, it queues it in a buffer called the *Outgoing capsules Buffer (OCB)* which subsequently causes the process called *capsule Write (CW)* to wake up in case it was sleeping. The role played by the CW is quite simple in the case of the View, Main and Admin Services. In these cases, the CW pops a capsule from the OCB and reads its *Propagation* field. If the latter reads *private*, the CW finds the right connector using the *Target connector ID* field in the capsule and writes the latter to the output stream of the connector. On the other hand, if the Propagation field reads that the capsule is in the public domain, then the CW writes that capsule to the output streams of all the connectors. The Data Service differs from the other Services at this point: if the Propagation field is public, the CW sends a ticket synchronization capsule to all the other coordinators via the Admin Services/Channels and it enters a wait mode. This is the part of the synchronization mechanism discussed above (see Sect. 3.5). After all the coordinators have acknowledged with their messages, the CW is waken up and the capsule that was previously removed from the OCB is distributed to all the connectors.

When the Services distribute capsules to all the connectors, the capsules are made private to prevent other coordinators from sending the capsules to their peers and hence cause redundancy. Most importantly, this prevents the situation

whereby the recipient coordinators in turn do not distribute the same capsule back and create a unending situation of pingpong effect. This is when capsules sent a coordinator are returned to the latter by the target ones and which will be sent again to them simply because the Propagation field of the capsule reads that it is meant to be distributed. Furthermore, the CW also makes sure that a capsule sent by a coordinator is only distributed to those connectors that are attached to participants because the originating coordinator has taken upon itself to send this capsule to all the other ones just like in this situation. Finally, the View Service differs from the others by the lack of a CA as it is not important for the coordinator to interprete capsules pertaining to updates of the View. So, the CR and CW are fused together in the sense that this new Process removes

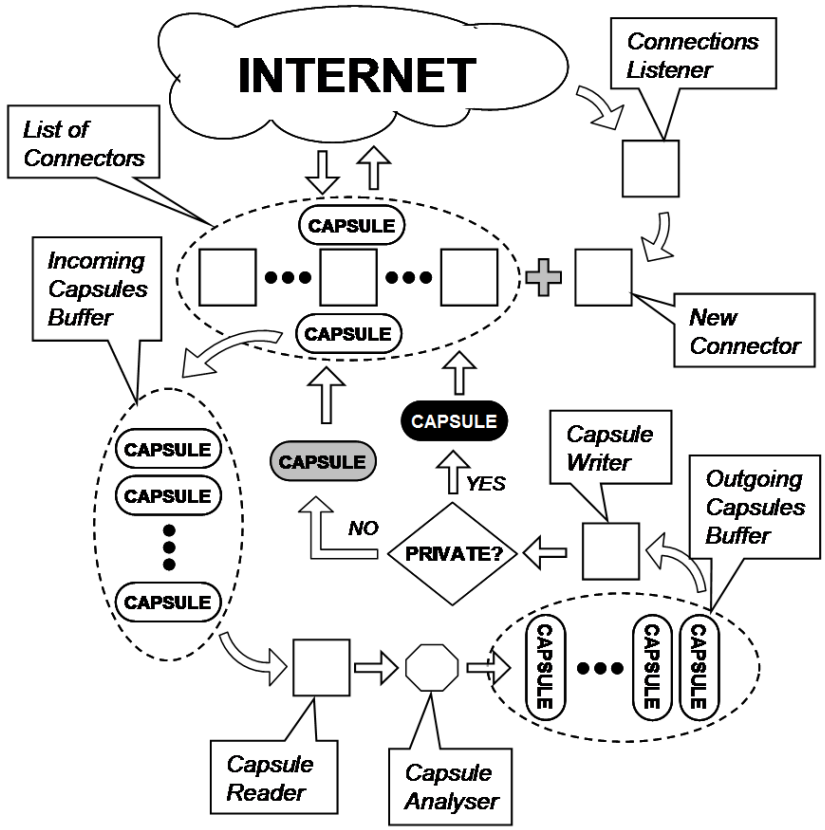

Figure 4.5 Composition of the Data Service in terms of Processes *(indicated by squares)* and the incoming and outgoing capsules. The dark coloured capsule indicates that it is Private and therefore intended for only one connector while the light grey one is meant to be distributed to *all* connectors.

capsules from the ICB and distributes it according to the above principle to all the connectors.

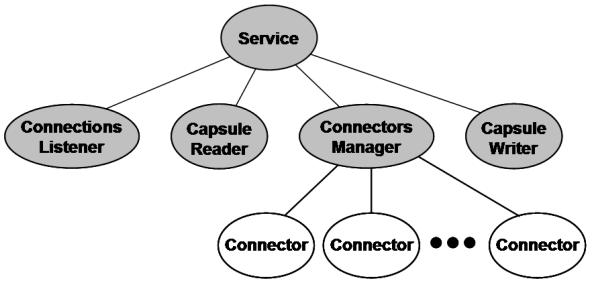

Figure 4.6 The dependency tree of the processes that make up a Service in a coordinator.

4.2.3 Distributed Model-View-Controller (DMVC)

The Model-View-Controller (MVC) design pattern is a flexible and robust architecture for designing applications. According to Helman [1998], the principle is to decompose an application into the following three components,

Model: Model is an encapsulation of the data part of the application and a set of methods used to query and update the state of the data. The concept of modelling in the design process serves to guide the focus onto finding the best approximation of the system to be implemented from the point of view of the states it can be in. This helps by isolating other worries, such as data representation, from the attention of the designers. Indeed, the model captures the essence the of the system by representing all the states and also by describing the relationships between these states.

View: View assists the design process by leading the aim of the exercise to finding out how the state of the system will be displayed to the user in terms of graphical and/or textual elements. In the end, the system will be used by humans and it is therefore crucial to get the interaction between the two parties right. Some applications deal with multi-dimensional data and it can be quite a challenging task to render the data into a meaningful form with limitations such as viewport size or the number of viewports.

Controller: Controller encapsulates the means by which a user can manipulate the model or update the view. In fact, the controller has the necessary

methods to capture the user's inputs via the hardware of choice, keyboard, mouse, etc. It then interpretes these events in the context of their occurence and call the appropriate methods in the model or the view. For example, if the user clicked the mouse on a *menu*, the controller invokes the methods in the view to drop down that menu. On the other hand, if the user filled in a textfield and clicked on an *OK* button, then the controller calls a method in the model to save this information.

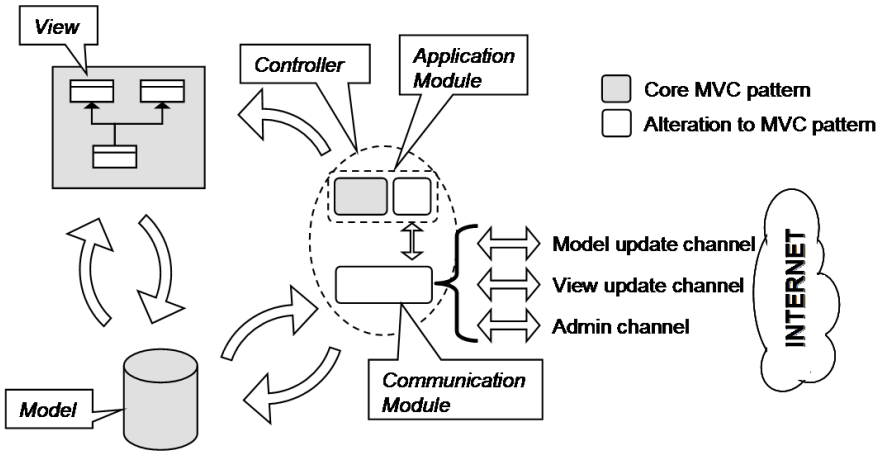

Figure 4.7 Modifications to the controller part of the MVC design pattern for building collaborative applications. The darker parts represent the original MVC design.

In order to integrate the MVC approach in the design of collaborative applications, it was necessary to ammend some modifications as illustrated in Fig. 4.7. The controller part has been extended to accomodate a communication module and the native methods that control the model. The view has been separated into two parts. The part that interacts with the communication module consists of those former native methods that are now made distributed by wrapping them with a special signature as shown in Table 4.2. (If the action of dragging an object on the viewport become distributive, then if a user drags something then everybody will see that object moving on their respective viewports.)

As illustrated in Fig. 4.8, the communication module is made up of a number of Processes and a CA. As before (see Sect. 4.2.2), the connector reads capsules from the Internet and adds them to the ICB while waiting to be processed. In this case, the CR and CW are fused into one process which hands over capsules

```
void setValue(int val) {
    value = val;
}
...
void setValue(String val) throws CTKException {
    try {
        value = Integer.parseInt(val);
    }
    catch(NumberFormatException e) {
        throw new CTKException("Conversion Error");
    }
}
```

Table 4.2: An example of a simple method overloaded to behave as a distributive method.

from the ICB to the CA, and writes any capsule returned by the latter to the connector. If the ICB is empty, the CR/CW Process enters a sleep mode waiting to be waken up when the connector adds a new capsule to it.

Fig. 4.9 illustrates the dependency tree of the processes that make up Communication Module. By the earlier discussion (see Sect. 4.1.1), we notice that the process responsible for updating the viewport (View Channel) is optional, thus is not a critical process and if it terminates abnormally, it will not bring down the participant.

4.2.4 Capsule Analyser

The Capsule Analyser (or analyser) is a processing unit closely tied to the CR process found in services and the Communication Module of participants. It is the part that makes remote execution possible in our collaboration model. When the capsule Reader process receives a capsule from the Internet, it hands it over to the analyser which prossesses the capsule according to the algorithm described in Table. 4.3 and wait for a result. Line *1* of the algorithm breaks down the capsule into the respective Command Objects by splitting the payload string about the command delimiters (see Table. 4.1). The resulting Command Objects

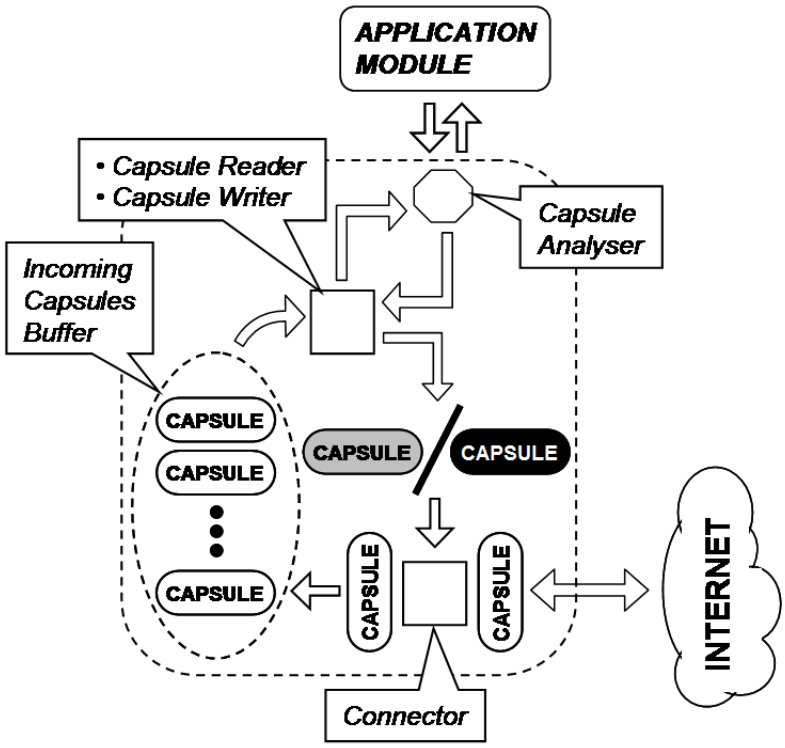

Figure 4.8 The inside of the communication module from the controller part of the DMVC design pattern.

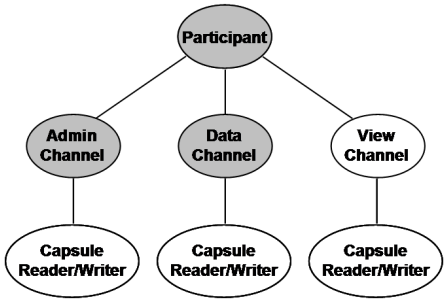

Figure 4.9 The dependency tree of the Processes that make up a the Communication Module of a participant.

are stored in an array which is scanned from start to end as indicated by the block of statements enclosed by lines *2* and *22*. The name of the remote object to be accessed, as well as the name of the method and a list of arguments to be used with the method are extracted from each Command Object. By line *8*, the remote

execution mechanism uses reflection to create an instance and load the required object into memory before it is acted upon by one of its method (see Sect. 2.1 for the reflection mechanism in Java). To reduce the huge amount of execution time associated to reflection, the objects that have already been loaded in the JRE have references to them stored in a hash table. This means that after the call to the object's method has ended, the object is prevented from being unloaded from the JVM by the garbage collector. So next time this object is needed, a direct reference to the object can be used instead of creating a new one. It is understood that the hash table can swell to enormous sizes if objects are only created and never garbage collected and this problem is currently being addressed even though the architecture works well for small sized applications. When a reference to the required object is obtained, as shown at line *8* or *11*, reflection is applied to make the dynamic call to the appropriate method embedded in that object. If the method call returns data, it will do so in terms of a Command Object and a reference to the latter is stored in the variable Resultcapsule as shown on lines *13* and *14*. Execution through Reflection can generate a number of Exceptions for a wide number of reasons. For example, if the name of the remote object is incorrectly specified then the Class Loader will not be able to find the definition of that object and thus throws a Runtime Exception. If an exception is thrown, it is caught and the Command Object currently being processed is tested to see if it is transactional by investigating the name of the called method. If this is not the case, the Exception is forgotten and the next Command Objects are treated as normally. On the other hand, the transactional Command Objects in the capsule that have sucessfully been executed so far are reprocessed. But this time the complementary *recover* methods of the formerly invoked methods are called to undo previous actions. The Resultcapsule is overwritten with an error Command Object with the caught exception. The processing of the capsule is finally interrupted. Depending on the flow of the program, at line *23*, Resultcapsule either has some command objects due to the successful processing of all the Command Objects from capsule or, due to an error as just explained, or it is empty. The latter case describes situations where a participant has sent a capsule to its coordinator or is from another coordinator and is meant to be distributed. In this case, the capsule analyser returns the original capsule, capsule to the capsule Reader process so that it can carry out the distribution to the capsule Reader process so that it can carry out the distribution. The situation where Resultcapsule is not empty and there have been no execution errors, describes the situation where a coordinator requests the participant to send some specific

information to it. For example, when the coordinator asks the participants to calculate *Responsiveness* and in which case the reply capsule from the them will be tagged private and the capsule analyser at the coordinator will return NULL as shown at line *28*.

4.3 Partial Replication

In our collaboration model, the software design of the participant is based around the MVC approach and therefore requires that the data (Model) is always an integral part of the collaborative application. Also, in the case of coordinators, it is functionally important they own their private copy of the data so that they can validate and authorize operations requested by the participants. In other terms, our collaboration model requires that data is fully replicated at all the participating sites (participants and coordinators). However, as discussed earlier (see Sect. 3.1) there are applications where the data is too big or a waste of space for it to be fully replicated. In these cases a partial replication of the data is best suited and our collaboration model can also adapt to this mechanism. If the data can be easily categorised in terms of usage groups, then for each group, a coordinator is made responsible for managing that part of the data. The interested users can join and leave at will the islands depending on their needs for the types of data. For example, based on the earlier situation about a collaborative UML editor (see Sect. 3.1), computers A and B could each have a coordinator that manages their respective portions of the data. This particular approach has a number of constraints, for example, a project manager can never view UML diagrams pertaining to the *Graphical User Interface (GUI)* and the database systems simultaneously. Moreover, when the project manager joins a new island, the data has to be downloaded first on its computer before any kind of collaboration can take place. But one of the motivation behind partial replication is that the data is too big to be kept in one place and therefore this would, for example, require the participant to offload old data before new one can be downloaded. This reinforces the limitation that the project manager can have only one view at a time. This problem can be resolved by using a light client approach to the design of the participant. With regards to the MVC pattern, the Model parts could be located only at the coordinators and when they are updated, they send capsules to refresh the viewport of the satellite participants. This thin client design can allow a participant to be a member of more than one island at a time and thus offers the possibility of many views on the multi-dimensional data.

```
1:  ArrayOfCmdObjects = decompose(capsule)
2:  For each CmdObj in ArrayOfCmdObjects
3:      Try
4:          ObjectName = CmdObj.getName()
5:          Method = CmdObj.getMethod()
6:          Arguments = CmdObj.getArguments()
7:          If [!HashTableOfAlreadyInvokedObjects.contains(ObjectName)]
8:              Obj = newInstance(ObjectName)
9:              HashTableOfAlreadyInvokedObjects.add(Obj)
10:         Else
11:             Obj = HashTableOfAlreadyInvokedObjects.getRef(ObjectName)
12:         End If
13:         ResultCmdObj = UseReflection(Obj, Method, Arguments)
14:         Resultcapsule.addCmdObj(ResultCmdObj)
15:     Catch(SomeException e)
16:         If [CmdObj.isTransactional()]
17:             Recover(ArrayOfCmdObjects, ObjectName)
18:             Resultcapsule.setCmdObj(ErrCmdObj, e.getMessage())
19:             Resultcapsule.setTarget(capsule.getSourceID())
20:             Break
21:         End If
22: End For
23: If [Resultcapsule.getNumOfCmdObjs() > 0]
24:     Return Resultcapsule
25: Else If [capsule.isDistributive()]
26:     Return capsule
27: Else
28:     Return NULL
29: End If
```

Table 4.3: The algorithm employed by the capsule analyser in the process of remote execution of operations.

However, this approach defeats one of the feature of our collaboration model where coordinators can be dynamically chosen from the pool of participants as they all have exactly the same model.

5 Experiments and Results

In this chapter, we give an empirical analysis of our collaborative model and we set to determine how many collaborators can an island support for a gui-based collaborative application. Therefore, it was decided to extend *FUJABA*[1](**F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain), a typical single-user application with a graphical interface, into one with multi-user capabilities, using CTK and our framework. FUJABA is a configurable suite of tools that can perform a wide spectrum of tasks, ranging from drawing UML behavioural diagrams, to carrying out design pattern recognition. This is possible because FUJABA's architecture is centered around a core that can be enhanced by pluging around it these various tools, thereby adding to it these particular characteristics. One of the basic features offered by FUJABA is the ability to draw UML class diagrams. This is the feature that we aim to extend into a collaborative task. There are two reasons why this feature was selected for the experiment.

Firstly, it comprises of a large amount of user-application interactions that is common to many applications. For example, many graphical applications provide interactions by allowing the users to drag items on the screen. Under a collaborative mode, it is a good idea to distribute these actions to all the collaborators, as it provides a sense of presence to the others. That is, every collaborator can observe the behavioural patterns of their peers through these broadcasted actions. It can be also argued that if all collaborators start to simultaneously drag a different item on the screen and these actions are replicated to each of them, then we are going to end up with a chaotic visual feedback at each collaborative terminal. To resolve this problem in FUJABA, we provide the ability for the user to switch on and off the subscription to the view update service at the coordinator.

Secondly, UML class drawing in FUJABA involves various types operations that are characterized by their frequency of usage. In other terms, there are operations that can be considered as slow, medium, or fast, based on the number of messages that they generate in a collaborative session. For example, when a user is editing the name of a class, this process can take several seconds depending on the typing speed of the latter and only one capsule is sent to the coordinator to validate that change. On the hand, if the user is changing the layout of the diagram by dragging the classes on the canvas, then several capsules describing the intermediary locations of the class item, have to be sent from the moment the item is picked off the canvas to that it is released at its final position. Here,

[1] `http://wwwcs.upb.de/cs/fujaba/`

it is necessary to send several capsules per second so that the motion of the class item on the canvases of the other collaborators appears as smooth as possible. Indeed, these last two operations are considered to be relatively slow and fast respectively. One must note that in the context of a different application, an operation that is considered fast could generate far fewer capsules than in the present one. But as already mentioned, FUJABA serves as a good test ground for analysing a wide variety of applications from an empirical point of view and in the context of our collaborative model.

When FUJABA is run with our collaborative plugin, we obtain the graphical interface illustrated in Fig. 5.1. The contribution of the plugin to the GUI is the panel of three *leds*, labelled as *Service Connections*. From left to right, the leds are red, green and blue in colour and they are each bound to a Channel linking them to the Admin, Data and View Services at the coordinator. The leds flicker at the rate of incoming traffic from the coordinator along their respective Channels and thus give a visual feedback about the running of the collaborative system. Only the blue led is made clickable and it allows the user to turn off and on the connection to the View Update Service when desired. From the plugin properties in Fig. 5.1, the Activate coordinator checkbox is used to indicate whether this FUJABA terminal will also serve as the primary coordinator of an island. In this case, only the Port Number field is used to create the coordinator part of the FUJABA application, while the participant part connects to the latter on that port and uses the host address 127.0.0.1.

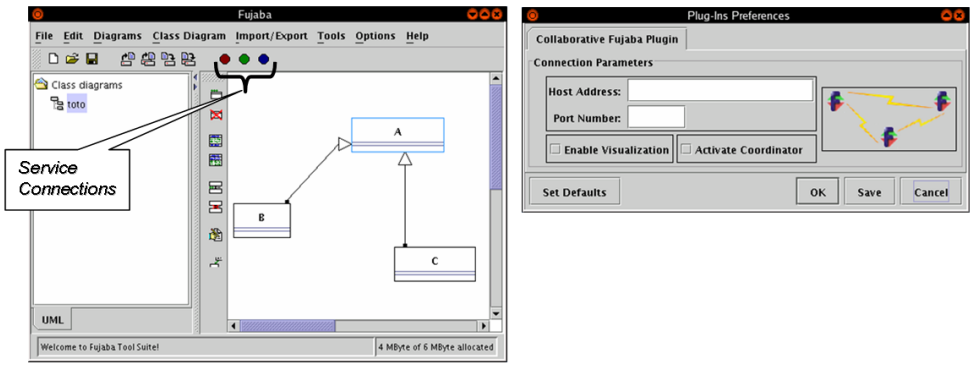

Figure 5.1 The picture on the left is FUJABA operating under the collaborative mode and on the right is the properties interface of the loaded plugin.

Next, the modified FUJABA application was operated alone in an island in order investigate the range of the rates of generation of capsules initiated by various operations, such as editing the name of the classes or dragging the class

objects on the canvas. After a number of runs and measurements, it was decided that an operation is considered to be very fast if it generates on average 20 capsules per second. This occured when class objects were dragged as violently as possible all over the canvas. It must be stressed that this is an extreme behaviour and therefore happens very rarely. However, this value serves as a very good upper limit to put the collaborative system under stress conditions.

For the actual experiment, we measured the size of the queue of capsules waiting to be processed at the coordinator. The reasoning behind this is that, the bigger the queue, the longer it takes for the request in a newly arrived capsule to be attended. Therefore, this experiment helps to determine how big an island can be (in terms of participants) under this stress condition so that the collaborative experience can still be considered as a reasonable one. The experiment also provides us with some insights on the strategies to use when deciding when to fragment an island into smaller islands. The test island consists of a Primary coordinator running on a $1Ghz$ Pentium computer with $256Mb$ of RAM memory and a Java heap of $512Mb$, and a varying number of automated participants, all located at network reachability in the range of $3ms$ to $4ms$. This means that if a participant sends a capsule to the coordinator on the network, it takes between $3ms$ to $4ms$ to get to its destination. For each run of the experiment, new test participants are added to the collaborative session and are configured to send capsules at the rate of 20 per second and for a duration of four minutes. In all these runs, the capsules are identical. They consist of one Command Object, meant to simulate a user dragging a class item on the canvas from one coordinate to another.

Fig. 5.2 illustrates how the queue at the coordinator grows as time passes by and as the number of collaborators increases. The graph only shows the runs with at least11 users, because before that the queue did not grow hardly at all. With 11 participants the coordinator manages to process all the buffered operations well within the four minutes period and it appears that afterwards no more major queues are formed. By contrasts, in the following run with 12 users, the coordinator also manages to purge the queue but it takes considerably longer time that with 11 participants. In the next run, the queue size stabilizes to a very large value (about 2000) which means that if a participant sends a request at this stage, there will be about 2000 operations ahead of it. This implies that the performance of the collaborative system has degraded. This is further confirmed by the following run with 14 users where the queue just keeps growing. Indeed, this experiment demonstrates that under such harsh conditions of 20

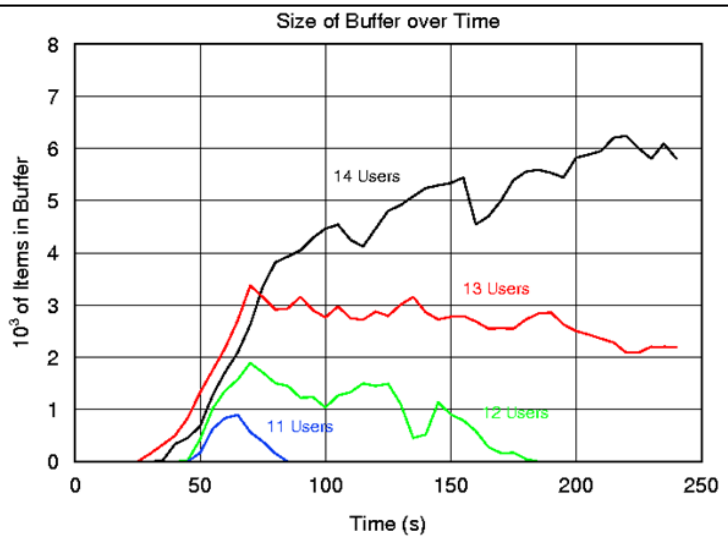

Figure 5.2 Each curve represents a run with a specific number (11- 14) of participants. Each participant generates 20 capsules per second.

capsules per second, and under the given hardware and software environment, a collaborative session can easily sustain 10 collaborators. Furthermore, if one more collaborator joins the session, the coordinator should start thinking about whether it is necessary to fragment the island into two smallers ones.

The experiment was repeated but this time the participants were configured to send only 2 capsules per second in order to observe how the model behave under more normal conditions. The results are illustrated in Fig. 5.3. The island can now cope with more than twice (roughly, 27 users) the number of participants than in the previous situation before the session becomes intolerable.

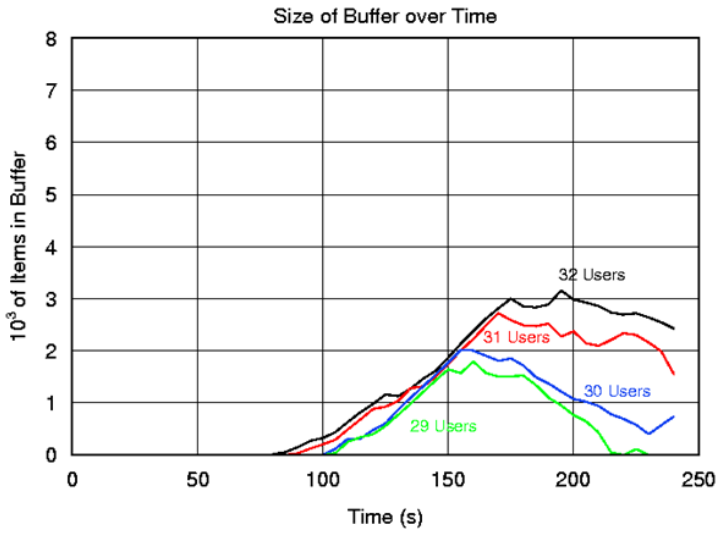

Figure 5.3 Each curve represents a run with a specific number (29-32)of participants. Each participant generates 2 capsules per second.

6 Conclusion

Since time immemorial, collaboration among communities, societies and countries has helped to raise civilizations, win wars, unite nations, drive science forward and bring technological advances to our lives. Therefore, it is only natural if we would like to collaborate in a virtual world in these times. The advent of the Internet has brought people closer together in the sense that nowadays, it takes only a matter of seconds to send a message from one corner of the world to another. Moreover, rapid technological innovations, such as video conferencing over IP, are helping to reduce the element of virtuality in this new environment. Collaborating on the Internet is nothing new though, as a matter of fact, it has been the motivational factor behind its creation. To date, the most famous collaborative application is undeniably the WWW, but one should acknowledge that there have been other applications such as email, or ftp prior to it. But since its inception, very few collaborative applications have been coming out on the Internet. The latest craze is file sharing through P2P networks, such Napster, and P2P computing, like for example the Seti@Home project. It seems that a clear crack is appearing between the communities of collaborative application developers and the consumer camps. In other terms, it looks like the middle group, those consumers enthusiastic about software development are dropping in numbers, either because the currently available softwares completely meet their needs, or the programming challenges are becoming too hard. Two of these concerns in the field of collaborative and distributed systems are, concurrent access under a controlled environment and the handling of excessive processing loads. These two issues are enough to discourage an individual from attempting to develop these applications. Or if they do, they tend to divert their effort from producing quality products.

To this end, we proposed in this thesis a framework and a toolkit that can be used to develop Java-based collaborative applications. It provides built-in mechanisms for concurrency control and load balancing. Concurrent accesses to data is regulated by the principle of mutual exclusion applied by a coordinator where only one operation can execute at a time while the others are made to wait. Although this strategy reduces considerably the advantage of parallelism, it guarantees the serializability of these operations. Moreover, the centralized point of serves as an ideal mechanism for distributing requests to the collaborators and conquently also ensures that they all receive the operations in the same order. The design of CTK also takes into consideration the single point of failure and

performance bottleneck of the central point that our collaborative topology suffers from. So, CTK offers a backup coordinator that kicks into action whenever the primary one dies and also provides a load balancing strategy based on dividing the topology into smaller versions called islands, thus distributing the overall processing load onto smaller units running in parallel. Consequently, CTK also implements a synchronization mechanism called a ticketing service that helps coordinate the execution of events happening simultaneously in all the islands. The actions of the coordinator are divided along four services offered to any participant which communicates with along Channels connected to each of these services. The Channel called Main is temporary and is used by a participant when it joins an island for the first time in a session. After that the participant communicates with coordinator using the Admin, View and Data Services. The Admin Service is used by the coordinator to gather statistical data from the participants of its islands. Presently, these data are only used to select appropriate Backup coordinators in an island. The Data Service is essentially used to send those operations that can violate the integrity of the system if it happens that they are executed in two different sequences at two sites. The View Service takes care of these operations where the order of these operations does not matter at all. For example, the operations generated by a user dragging an object on the screen are sent along the View Channel. From a software architecture point of view, the participant is built according to the MVC design pattern. This way, CTK can easily plug into the application through the Controller part. This makes it suitable for a multi-user environment. Moreover, CTK favours applications built using a connection-oriented paradigm and the computing units (coordinators and participants) make use of Java's reflection mechanism to process incoming capsules.

We also demonstrated in this thesis the effectiveness of the toolkit by using it to extend a single user application to multi-user capabilities and it worked fine. Moreover, through a series of experiments, we also discovered that in our collaborative model, islands can support around 10 participants if their operating speed is very fast and if they are within a network reachability of 3 to $4ms$. However, under a more relaxed operational mode, where 2 capsules are generated per second on average by each participant, then an island can easily accomodate around 25 users. This experiment also provided us with some insight on the threshold of the number of participants joining a session when an island has to be split into two smaller halves.

Appendix A - Threads in Java

In Java, a thread is defined as an execution context or a lightweight process [Sun, 2004d]. This characteristic enables several statements or expressions to be executed or evaluated concurrently in a multi-threaded environment like the JRE. Essentially, a thread can be in four states, two of which are terminal states. Fig. 6.1 illustrates these states and the method primitives that cause the transitions between them.

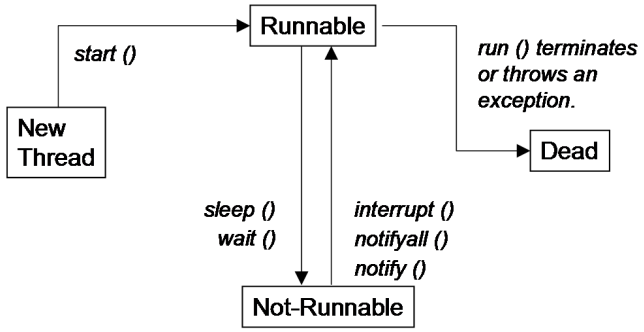

Figure 6.1 A state chart of the life cycle of a Java thread.

New thread: New threadstate is entered when a new thread instance is created. At this point, the thread object has only been initialized[1] with the necessary parameters so that it is ready to access the CPU whenever it is requested to do so. But, the thread does not know yet the statements it is going to execute.

Runnable: When start() is invoked on a thread, it enters Runnable state but it is not necessarily running yet as many other threads might have been started also at roughly the same time. Or it could have been started while some other thread was using the CPU to execute some statements. In this state, the thread acquires the necessary system resources such as virtual memory for its execution stack and enters a waiting queue for scheduled execution. When it gets its turn to access the CPU, it enters the run() block of statements and sequentially executes a maximum during its allocated time slot. After which the CPU is taken away from it and it remains in that state if it has still more statements to execute.

[1] From source code of java.lang.Thread.java of JDK 1.4.2

It must be stressed that there are two different implementations of threads, native and green which differ in many aspects. One of them is the scheduling of threads for execution. This implies that there is no guarantee that a group of threads would access the CPU in the same sequence if they are started at the same time. Therefore this can cause the same multithreaded application to produce different results when run on different JVMs.

Not-Runnable: There are two ways for a thread to enter Not-Runnable state, either by invoking the sleep() method or the wait() method on it. The thread can enter the runnable state any time one of the following methods are invoked: interrupt(), notify() or notifyall(). However, if sleep() was used, the thread can also return to the runnable state after the *sleeping* time has elapsed. Again, when the thread enters the runnable state it waits for its turn to access the CPU.

The interrupt() call is particularly useful in the design of turning Java threads into event sensitive processes. Every thread contains a status flag that is set when this method is invoked on the thread object. In the case, that the thread was in the not-runnable state, both methods sleep(), wait() throw a RuntimeException issued by the JRE. This exception must be caught or it will escape the block of statements and can cause trouble as described below in the section about *dead* state. A good compiler will halt the compilation process if the programmer has not wrapped a try-catch statement around these method calls. So, once the thread is interrupted in its *sleep* or *wait* mode, it is automatically in the *runnable* mode. On the other hand, if the thread was in the latter mode when interrupt() was invoked on the thread object, then nothing will happen. However, one can use either interrupted() or isInterrupted() to read the flag set by the interrupt() method call. So in situations where loops are used in the run() block of statements, it is a good idea to use these methods with a logical operator in the condition testing part of the loop block.

Dead: Dead state is an irreversible state unlike the previous two. A thread can be sent to that state in three different ways. The most natural way is when the last statement in the run() block has been executed. The second possibility is if a RuntimeException escapes the execution block. The third possiblity is if the reference to the thread is lost or set to NULL and in which case it will throw a similar exception. If a thread dies unnaturally, the results can be

catastrophic. For example, if the thread held locks on some objects, then those locks are never returned and therefore prevents other threads from accessing those objects. So, a good programming approach would be to catch all exceptions and throw new ones appropriately in order to maintain a tight control over the termination of a thread.

Finally there is one more method called isAlive() that is useful in our design goal. This method returns a boolean value to indicate in which of the two pairs of states the thread is in, (*New Thread* or *Dead*) or (*Runnable* or *Not-Runnable*). So, if it returns true, then it means that the thread is either in the *Runnable* or *Not-Runnable* state.

One main concern with threads is with concurrent execution. Consider a program with two threads T_1 and T_2 that operate on the same variable v. This situation can be non-problematic if the host computer has only one processor and the operation is relatively simple so that either T_1 or T_2 have enough time to complete during their first access to the CPU. However, if there is more than one processor involved and the threads need more than one access to the CPUs, then this could lead to a situation where end result of v is not stable. Suppose T_1 is accessing v on CPU_a and at the same time T_2 is executing on CPU_b, and they need many accesses before they can complete. Now if the program is run a number of times, there is no guarantee that the accesses of T_1 and T_2 on v will occur in exactly the same sequences all the time. So, it is very likely that v would be in different states each time the program is run. Fortunately, the Java Language Specification accomodates for many threads to execute concurrently on the same data though a mechanism built around mutual-exclusion. This requires that every object in Java inherit from a fundamental entity called java.lang.Object which holds a *monitor*. According to the Java Language specification [Gosling *et al.*, 1996], the roles of the monitor are:

- It accepts requests from threads to access the object that it is supervising. If the object is not readily available, because some other thread has exclusive access to it already, the requesting thread is added to a pool of waiting threads. At this point, that thread is said to have *entered the monitor*.

- When the object becomes available, it selects a thread from the pool and grant it access to that object. If at this point, the thread requires access to other objects, then a counter associated to the thread is incremented.

When a thread has been granted access to an object, it is said to *own the monitor*.

- The monitor moves to the waiting pool, those threads that have temporarily relinquished access to the monitored object. This happens when the wait() method of the object is invoked by the owning thread. Under this circumstance, the thread is said to have *released the monitor*.

- When a thread is done operating on an object, the latter's monitor decrements the monitor-owning counter associated to it and the thread is sent to the waiting pool. And when that counter reads zero, the monitor removes it from the pool and the thread is said to have *exited the monitor*.

The mutual exclusion mechanism arises from the fact that only one thread at a time can own the monitor. These roles of the monitor are translated at the application or programming level to *locking* and *unlocking* the monitored object. A thread enters an object's monitor by wrapping a reference to the object in a synchronise block. The monitor is exited when the last statement in that block has returned. Fig. 6.2 illustrates the intricate relationship between a thread and the monitor of an object.

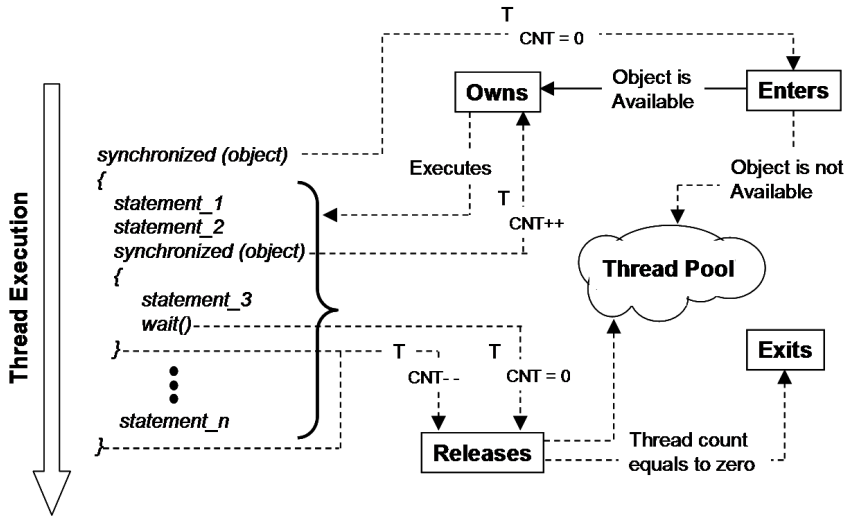

Figure 6.2 The state diagram of the monitor of an object as an thread passes through a synchronised block.

Finally, an object in Java possesses two more methods of interests to the CTK Process design, notify() and notifyAll() which have already been introduced

in Fig. 6.1. When a thread invokes any of these methods on an object, it actually indicates to the object's monitor that it is releasing control. If the method used was notifyAll(), the monitor wakes all the other threads that are waiting on its object so that they then compete to own the monitor. On the other hand, if the method used was notify(), the monitor selects one of the waiting threads and grants it control of the object. wait() and notify() are two useful methods that can be used to solve the producer-consumer problem common to many synchronous collaborative and distributed applications. In this situation, the producer and consumer are two threads that have synchronised access on a object which acts as a place-holder for the particular item of production and consumption. The consumer thread invokes the wait() method on the object whenever it is empty and when the producer thread calls the notify() method on that object after it puts some data in it. That way, the monitor wakes the consumer thread which process that data.

Bibliography

[Abbate, 2000] Janet Abbate. *Inventing The Internet*. MIT Press, 2000.

[Allamaraju *et al.*, 2001] Subrahmanyam Allamaraju, Karl Avedal, Richard Browett, J̃ason Diamond, John Griffin, Mac Holden, Andrew Hoskinson, R̃od Johnson, Tracie Karsjens, Larry Kim, Andrew Longshaw, T̃om Myers, Alexander Nakhimovsky, Daniel O'Connor, Sameer Tyagi Geert Van Damme, Gordon van Huizen, Mark Wilcox, and Stefan Zeiger. *Profession Java Server Programming J2EE Edition*. Wrox Press, 2001.

[Authority, 2004] Internet Assigned Numbers Authority. Historical development of p2p networks. Website, Apr 2004. `http://www.iana.org/assignments/port-numbers`.

[Balakrishnan *et al.*, 2003] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2), 2003.

[Bourke, 2001] Tony Bourke. *Server Load Balancing*. O'Reilly and Associates, Inc., 2001.

[Brinck, 1998] Tom Brinck. Groupware: Applications. Website, 1998. `http://www.usabilityfirst.com/groupware/applications.txl`.

[Brosnan *et al.*, 2002] Andrew Brosnan, Tarun Maitrat, Andrew Colhoun, and Bob MacArdle. Historical development of p2p networks. Website, 2002. `http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p1.html`.

[Cass and Kushner, 2002] Stephen Cass and David Kushner. The wizardry of id. Website, Aug 2002. `http://www.spectrum.ieee.org/WEBONLY/publicfeature/aug02/id.html`.

[Chan *et al.*, 1998] Patrick Chan, Rosanna Lee, and Doug Kramer. *The Java Class Libraries*. Addison Wesley Longman, Inc., 1998.

[Chow and Jonhson, 1998] Randy Chow and Theodore Jonhson. *Distributed Operating Systems and Algorithms*. Addison Wesley Longman, Inc., 1998.

[Collins, 2005] William Collins. *Data Structures and the Java Collections Framework*. Mc Graw-Hill International Edition, 2005.

[Comer, 1991] Douglas E. Comer. *Internetworking With TCP/IP - Vol 1: Principles, Protocols and Architecture*. Prentice Hall, Inc., 1991.

[Coulouris *et al.*, 2001] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*. Addison Wesley Longman, Inc., 2001.

[Deering, 1989] S. Deering. Hosts extension for ip multicasting. Website, 1989. `http://www.faqs.org/rfcs/rfc1112.html`.

[Farley, 1998] Jim Farley. *Java Distributed Computing*. O'Reilly and Associates, Inc., 1998.

[Feibel, 1996] Werner Feibel. *The Encyclopedia of Networking*. Network Press, 1996.

[Florin and Toinard, 1992] G. Florin and C. Toinard. A new way to design causally and totally ordered multicast protocols. *ACM SIGOPS Operating Systems Review*, 26(4), Oct 1992.

[Gosling *et al.*, 1996] Jame Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley Longman, Inc., 1996.

[Green, 2004a] Dale Green. Java tutorial trail: The reflection api. Website, 2004. `http://java.sun.com/docs/books/tutorial/reflect/`.

[Green, 2004b] Roedy Green. Java glossary : endian. Website, 2004. `http://mindprod.com/jgloss/endian.html`.

[Grosso, 2001] William Grosso. *Java RMI*. O'Reilly and Associates, Inc., 2001.

[Hapner *et al.*, 2002] Mark Hapner, Richard Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service specification - version 1.1. Website, 2002. `http://java.sun.com/products/jms/docs.html`.

[Harold, 2000] Elliotte Rusty Harold. *Java Network Programming*. O'Reilly and Associates, Inc., 2000.

[Hayes, 1998] Brian Hayes. Collective wisdom. *American Scientist*, 86(2), Mar 1998.

[Held, 2002] Gilbert Held. *Ethernet Networks*. John Wiley and Sons, Ltd., 2002.

[Helman, 1998] Dean Helman. Model-view-controller (mvc). Website, 1998. `http://ootips.org/mvc-pattern.html`.

[Hunter and Crawford, 2000] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly and Associates, Inc., 2000.

[Information Sciences Institute, 1981] University of Southern California Information Sciences Institute. Rfc 791: Internet protocol. Website, 1981. `http://www.ietf.org/rfc/rfc791.txt`.

[Intelligraphics, 2004] Intelligraphics. Ip-multicasting technology. Website, 2004. `http://www.intelligraphics.com/articles/ipmulticasting1_article.html`.

[Joyce and Moon, 2000] Jerry Joyce and Marianne Moon. *Microsoft Windows 2000 Professional*. Microsoft Press, 2000.

[Lamport, 1978] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), Jul 1978.

[Lamport, 1990] Leslie Lamport. Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems*, 8(4), Nov 1990.

[Leiner *et al.*, 2003] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. A brief history of the Internet. Website, 2003. `http://www.isoc.org/internet/history/brief.shtml`.

[Liang and Bracha, 1998] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[Naughton and Schildt, 1999] Patrick Naughton and Herbert Schildt. *Java 2: The Complete Reference*. McGraw-Hill International Edition, 1999.

[Naulls, 2004] Peter Naulls. C in risc os. Website, 2004. http://www.riscos.info/riscos/.

[Nuckchady and Nummenmaa, 2003] Y.C. Nuckchady and J. Nummenmaa. An architecture for building collaborative applications in java. *Proceedings of the 8th Symposium on Programming Languages and Software Tools*, 2003.

[Oaks, 2001] Scott Oaks. *Java Security*. O'Reilly and Associates, Inc., 2001.

[Oosthoek, 1997] Simon Oosthoek. Survey of multicast support for ip over atm for implementation purposes. Master's thesis, Department of Computer Science, University of Twente, http://margo.student.utwente.nl/simon/finished/thesis/, July 1997.

[Orfali *et al.*, 1999] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client/Server Survival Guide*. John Wiley & Sons Inc., 1999.

[Perrone and Chaganti, 2000] Paul J. Perrone and Venkata S. R. 'Krishna' R. Chaganti. *Building Java Enterprise Systems with J2EE*. Sams Publishings, 2000.

[Rodley, 1996] John Rodley. *Writing Java Applets*. Coriolis Group Books, 1996.

[Sun, 1999] Microsystems Sun. Extension mechanism architecture. Website, 1999. http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html.

[Sun, 2002] Microsystems Sun. Java cryptography architecture (jca). Website, 2002. http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html.

[Sun, 2003] Microsystems Sun. Java remote method invocation. Website, 2003. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[Sun, 2004a] Microsystems Sun. Java cryptography extension (jce). Website, 2004. http://java.sun.com/products/jce/.

[Sun, 2004b] Microsystems Sun. Java secure socket extension (jsse). Website, 2004. http://java.sun.com/products/jsse/.

[Sun, 2004c] Microsystems Sun. The java tutorial - the life cycle of an object. Website, 2004. http://java.sun.com/docs/books/tutorial/java/data/objects.html.

[Sun, 2004d] Microsystems Sun. What is a thread? Website, 2004. http://java.sun.com/docs/books/tutorial/essential/threads/definition.ht%ml.

[Systems, 2003] Cisco Systems. Internetworking technology handbook. Website, 2003. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/.

[Tanenbaum and van Steen, 2002] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems - Principles and Paradigms*. Prentice-Hall, Inc., 2002.

[Venners, 1998] Bill Venners. *Inside the Java Virtual Machine*. McGraw Hill, 1998.

[Wikipedia, 2004] Wikipedia. History of the Internet. Website, Mar 2004. http://en.wikipedia.org/wiki/History_of_the_Internet.